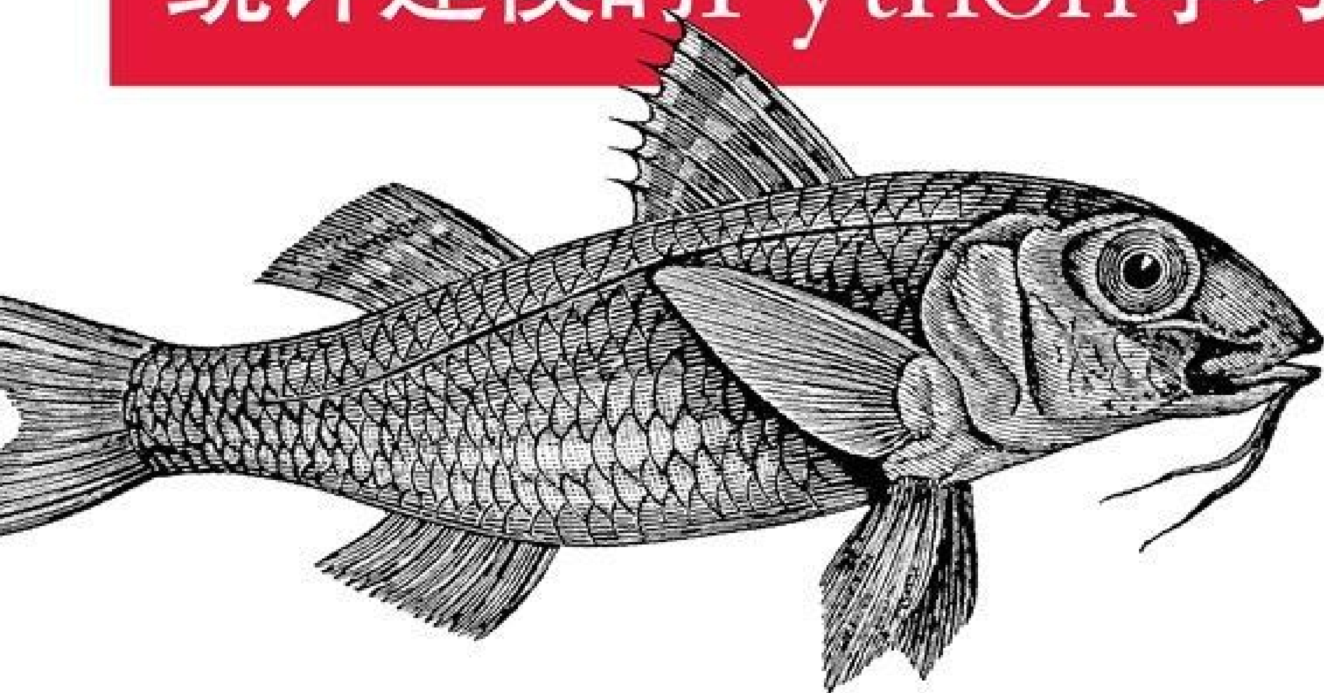
Think Bayes

贝叶斯思维：
统计建模的Python学习法



[美] *Allen B. Downey* 著
许杨毅 译

O'REILLY®

人民邮电出版社
POSTS & TELECOM PRESS

目　录

版权信息

书名：贝叶斯思维：统计建模的Python学习法

ISBN：978-7-115-38428-7

- 著　[美] Allen B. Downey

  译　许杨毅

  责任编辑　王峰松

- 人民邮电出版社出版发行　北京市丰台区成寿寺路11号

  邮编　100164　电子邮件　315@ptpress.com.cn

  网址　http://www.ptpress.com.cn

- 读者服务热线：(010)81055410

  反盗版热线：(010)81055315

版权声明

内容提要

这本书旨在帮助那些希望用数学工具解决实际问题的人们，仅有的要求可能就是懂一点概率知识和程序设计。贝叶斯方法是一种常见的利用概率学知识去解决不确定性问题的数学方法，对于一个计算机专业人士，应当熟悉其在诸如机器翻译、语音识别、垃圾邮件检测等常见的计算机领域的应用。

本书实际上会扩大你的视野，即使不是一个计算机专业人士，你也可以看到在战争环境下（第二次世界大战德军坦克问题），法律问题上（肾肿瘤的假设验证），体育博彩领域中（棕熊队和加人队NHL比赛问题）贝叶斯方法的威力。怎么从有限的信息判断德军装甲部队的规模？你所支持的球队有多大可能赢得冠军？在《龙与地下城》勇士中，你应当对游戏角色属性的最大值有怎样的预期？甚至在普通的彩弹射击游戏中，拥有一些贝叶斯思维也能帮助你提高游戏水平。

除此以外，本书在共计15章的篇幅中讨论了怎样解决十几个现实生活中的实际问题。在这些问题的解决过程中，作者还潜移默化地帮助读者形成了建模决策的方法论，建模误差和数值误差怎么取舍，怎样为具体问题建立数学模型，如何抓住问题中的主要矛盾（模型中的关键参数），再一步一步地优化或者验证模型的有效性或者局限性。在这个意义上，这本书又是一本关于数学建模的成功样本。

推荐序

很多人把世界理解为基于简单的、确定的，非一即零、非黑即白的。但是真实的世界却是非常复杂的，不是一两个公式可以完美总结概括的。就像我们的高考成绩和我们的学习水平，确实有很大的联系，但是最后又会受到很多因素的影响（比如身体状况，是否休息好了，心情，天气等），进而使得我们的最终成绩在真实水平上下有很大的波动。这就像我们分析很多事情时，经常得到的结论，“既有必然性，又有偶然性”。

这个时候，基于概率和统计的方法给了我们很多的帮助。很多时候，我们不能给出每一个人、每一件事的确定结果。但是当我们观察大量的相同事件后，我们就会发现从一个集体的意义上的规律是存在的。而单个事件每次可能得到不同的结果，这些结果以最有可能的结果为中心，服从一定的概率分布。了解这些分布数据，使我们更加容易理解和预期真实世界的多边形。

回顾在进行计算机自然语言处理过程中走过的路，我们就会发现从研究规则到研究统计的转变。最初，研究人员都认为，语言是基于语法规则。这个也很容易理解，因为我们学习语言的时候，总是背单词，学语法，然后掌握语言。基于这种思维，自然语言处理经历了多年的发展后，遇到了巨大的挑战。那就是即便语法规则已经非常复杂，仍然不能处理大多数的语言情况。从结果上而言，自然语言处理的准确度远低于人类，不具有真正的使用价值。而后，有一批学者开始另辟蹊径，基于统计的思路进行探索。如果语言是根据人类沟通需求自然发生，然后才有总结出来的语法呢？基于这种思想，研究人员放弃语法规则，开始建立基于统计的模型。他们使用了大量的真实文本数据，分析每个词和它前后的词出现的统计关系，用贝叶斯方法以及马尔科夫过程，建立了新的自然语言处理模型。这一次，语言处理准确率有了巨大的提升，进而达到可以实用的要求。今天，当我们使用谷歌翻译、苹果的Siri语音服务的时候，后面都有基于统计的模型的功劳。

还有很多的真实世界的事情都是这样的，比如路上的交通是否阻塞、银行排队的时间、球赛的比赛结果，都是以一种概率的形式出现的。了解贝叶斯方法，也是了解真实世界运行的一种有效途径。本书中

也列举了很多的真实实例来告诉我们，贝叶斯方法和真实世界的联系。

另外，在我们正在经历的大数据时代，作为数据分析方法的一个巨大分支，基于贝叶斯的机器学习算法也在被广泛地使用，并产生很多实际意义。比如简单贝叶斯算法、贝叶斯信念网络等，被广泛地应用于分类和预测。对于海量数据的文本分类问题，例如，垃圾邮件的甄选和过滤，基于贝叶斯方法的算法取得了非常好的效果，并在很多公司中正在使用，帮助我们远离垃圾邮件的骚扰。

更加难能可贵的是，本书作者用相对简单的Python语言，对所涉及的实例进行了编程。对于有一定计算机基础的人来说，通过程序，可以进一步理解贝叶斯方法的应用，真正掌握并且可以利用这些程序达到举一反三的效果。

本书用简洁的语言，大量的实例和故事，辅之以简单的Python语言，把原本枯燥的概率理论讲得生动且容易理解。在学习到理论的同时，还了解了它的真实意义以及可以使用的地方。对于有追求的工程师和感兴趣的读者而言，这是一本提升自我的很好的图书。

酷我音乐　雷鸣 [①]

---

① 雷鸣，现任酷我音乐董事长、CEO，国家千人计划特聘专家，百度创始七剑客之一，百度搜索引擎的早期设计者和技术负责人之一。获北京大学计算机科学硕士学位和斯坦福大学商学院MBA学位，曾任北京大学计算机系学生会主席和斯坦福大学中国学生学者联合会副主席。

前言

学习之道

这本书以及Think系列其他书籍的一个前提是：只要懂得编程，你就能用这个技能去学习其他的内容 。

绝大多数贝叶斯统计的书使用数学符号并以数学概念的形式表示数学思想，比如微积分。但本书使用了Python代码而不是数学，离散近似而不是连续数学。结果就是原本需要积分的地方变成了求和，概率分布的大多数操作变成了简单的循环。

我认为这样的表述是易于理解的，至少对于有编程经验的人们来说是这样的。当作建模选择时也非常实用，因为我们可以选取最合适的模型而不用担心偏离常规分析太多。

另外，这也提供了一个从简化模型到真实问题的平滑发展路线，第3章就是一个好示例。它由一个关于骰子的简单例子开始，那是基本概率的一个主题；紧接着谈到了一个我从Mosteller《50个挑战的统计学难题》（Fifty Challenging Problems in Probability）一书中借用的火车头问题；最后是德军坦克问题，这个第二次世界大战中成功的贝叶斯方法应用案例。

建模和近似

本书中多数章节的灵感都是由真实世界里的问题所激发的，所以涉及了一些建模知识，在应用贝叶斯方法（或者其他的分析方法）前，我们必须决定真实世界中的哪些部分可以被包括进模型，而哪些细节可以被抽象掉。

例如，第7章中那个预测冰球比赛获胜队伍的例子，我将进球得分建模为一个泊松过程，这预示着在比赛的任何时段进球机会都是相等的，这并不完全符合实际情况，但就大多数目的来说可能就够了。

第12章中，问题是对SAT得分进行解释（SAT是用于全美大学的入学标准测试）。我以一个假设所有SAT试题难度相同的简化模型开始，但其实SAT的试题设计中既包括了相对容易，也包括了相对较难的试题。随后提出了第二个反映这一设计目的的模型，结果显出两个模型在最终效果上没有大的差别。

我认为在解决问题的过程中，明确建模过程作为其中一部分是重要的，因为这会提醒我们考虑建模误差（也就是建模当中简化和假设带来的误差）。

本书中的很多方法都基于离散分布，这让一些人担心数值误差，但对于真实世界的问题，数值误差几乎从来都小于建模误差。

再者，离散方法总能允许较好的建模选择，我宁愿要一个近似的良好的模型也不要一个精确但却糟糕的模型。

从另一个角度看，连续方法常在性能上有优势，比如能以常数时间复杂度的解法替换掉线性或者平方时间复杂度的解法。

总的来说，我推荐这些步骤的一个通用流程如下。

1．当研究问题时，以一个简化模型开始，并以清晰、好理解、实证无误的代码实现它。注意力集中在好的建模决策而不是优化上。

2．一旦简化模型有效，再找到最大的错误来源。这可能需要增加离散近似过程当中值的数量，或者增加蒙特卡洛方法中的迭代次数，或者增加模型细节。

3．如果对你的应用而言性能就已经足够了，则没必要再优化。但如果要做，有两个方向可以考虑：评估你的代码以寻找优化空间，例如，如果你缓存了前面的计算结果，你也许能避免重复冗余的计算；或者可以去发现找到计算捷径的分析方法。

这一流程的好处是第一、第二步较快，所以你能在投入大量精力前研究多个可替代的模型。

另一个好处是在第三步，你可以从一个大体正确的可参考实现开始进行回归测试。也就是，检查优化后的代码是否得到了同样的结果，至

少是近似的结果。

代码指南

本书中的很多例子使用了在thinkbayes.py 当中定义的类和函数，可以从 *http://thinkbayes.com/thinkbayes.py* 下载这个模块。

本书大多数章节包括了可以从 *http://thinkbayes.com* 下载的代码，其中有一些依赖代码也需要下载，我建议你将这些文件全部放入同一个目录，这样代码间就可以彼此引用而无需变更Python的库文件搜索路径。

你可以在需要时再下载这些代码，或者一次性从 *http://thinkbayes.com/thinkbayes_code.zip* 下载，这个文件也包括了某些程序使用的数据文件，当解压时，将创建名为thinkbayes_code 的包括本书中所有代码的目录。

另外，如果是Git用户，你可以通过fork和clone来一次性获得这个仓库： *https://github.com/AllenDowney/ThinkBayes* 。

我用到的模块之一是thinkplot.py ，它对pyplot 中一些函数进行了封装，要使用它需要安装好matplotlib ，如果还没有，检查你的软件包管理器看看它是否存在，否则你可以从 *http://matplotlib.org* 得到下载指南。

最后，本书中一些程序使用了NumPy和SciPy，可以从 http://numpy.org 和http://scipy.org 获得。

编码风格

有经验的Python程序员会注意到本书中的代码没有符合PEP 8这一最通用的Python编码指南（ *http://www.python.org/dev/peps/pep-0008/* ）。

确切地说，PEP 8使用带有词间下划线的小写函数名like_this，而在本书中和实现的代码里，函数和方法名以大写开头并使用间隔式的大小写，LikeThis。

没有遵循PEP 8规范的原因是在我为书中内容准备代码时正在谷歌做访问学者，所以就遵循了谷歌的编码规范，它只在少数地方沿袭了PEP 8，一用上了谷歌风格我就喜欢上了，现在要改太麻烦。

同样，在主题风格上，如在“Bayes’s theorem”中，s放在单引号后，在某些风格指南中倾向这样使用而在其他指南当中不是。我没有特别的偏好，但不得不选择其一，所以就是你们现在看到的这个。

最后一个排版上的注脚是：贯穿全书，我使用PMF和CDF表示概率密度函数或累积分布函数这些数学概念，而Pmf和Cdf是指我所表述的Python对象。

预备条件

还有几个出色的能在Python中进行贝叶斯统计的模块，包括pymc和OpenBUGS，由于读者需要有相当多的背景知识才能开始使用这些模块，因此本书中我没有使用它们，而且我想使阅读本书的预备条件最小。如果你了解Python和一点点概率知识，就可以开始阅读本书。

第1章关于概率论和贝叶斯定理，没有程序代码。第2章介绍了Pmf，一望而知是用来表示概率密度函数（PMF）的Python字典对象。然后第3章我介绍了Suite，一个Pmf对象，也是一个能进行贝叶斯更新的框架，因而万事具备了。

好了，随后的章节中，我使用了高斯（正态）分布，二次和泊松分布，beta分布等各种分析型的概率分布，在第15章，我介绍了不太常见的狄利克雷分布，不过接着也进行了解释。如果你不熟悉这类分布，可以从维基百科了解它们。也可以阅读本书的一本指南《统计思维》（Think Stats），或其他入门级的统计学书籍（不过，恐怕大多数类似书籍都会采取对实战没有太大帮助的数学方法来阐述）。

书中使用的惯例写法

本书中使用了下面的印刷惯例。

斜体（*Italic* ）

表示新术语，URL，邮件地址，文件名和文件扩展名。

等宽（`Constant width`）

用于程序代码，也包括那些表示程序代码元素的段落，例如，变量和函数名，数据库，数据类型，环境变量，声明和关键字。

等宽粗体（**`Constant width bold`**）

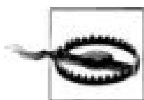
命令或者其他由用户输入的文字。

等宽斜体（*`Constant width italic`*）

应该由用户输入值替换或者由上下文决定的文本。



这个图标表示这是一个提示、建议或者一般性的注记。



这个图标表示这是提醒或者警示。

我们的联系方式

如果你想就本书发表评论或有任何疑问，敬请联系出版社。

美国：

O’Reilly Media Inc.

1005 Gravenstein Highway North

Sebastopol， CA 95472

中国：

北京市西城区西直门南大街2号成铭大厦C座807室（100035）

奥莱利技术咨询（北京）有限公司

我们还为本书建立了一个网页，其中包含了勘误表、示例和其他额外的信息。你可以通过地址访问该网页： *http://oreil.ly/think-bayes* 。

关于本书的技术性问题或建议，请发邮件到：bookquestions@oreilly.com。

欢迎登录我们的网站（ *http://www.oreilly.com* ），查看更多我们的书籍、课程、会议和最新动态等信息。

我们的其他联系方式如下。

Facebook： *http://facebook.com/oreilly*

Twitter： *http://twitter.com/oreillymedia*

YouTube： *http://www.youtube.com/oreillymedia*

贡献者列表

如果你发现本书有需要更正的地方或者其他建议，请发送电子邮件至downey@allendowney.com。一旦根据你的反馈进行了修正，我会将你加入贡献者列表（除了要求不署名的情况）。

提供包含错误之处的段落部分，会让我更容易找到它们。只提供页和节数也可以，但还是不太容易找到错误之处。这里先致谢！

- 首先，我要感谢大卫·麦凯（David MacKay）的优秀作品《信息理论、推理和学习算法》（Information Theory, Inference, and Learning Algorithms），我从这本书里第一次理解了贝叶斯方法。他允许我使用他书中的几个问题来作为例子。
- 这本书也得益于我和圣乔恩·马哈的相互配合，2012的秋天我在欧林学院审核了他的贝叶斯推理课程。
- 在参加波士顿Python用户组项目时，我在夜班时间完成了本书的部分内容，所以我也要感谢他们以及他们所提供的比萨。

- 乔纳森·爱德华兹提交了第一个拼写错误。
- 乔治·珀金斯发现了一个标记错误。
- 奥利维尔提出了几个有益的建议。
- 尤里·帕西奇尼克发现了几个错误。
- 克里斯托弗·欧霍特提交了一个更正和建议的清单。
- 罗伯特·马库斯发现了一个错误放置的小写i。
- 麦克斯·黑尔珀林建议在第1章提供一个澄清章节。
- 马库斯·杜布勒指出“从碗中有放回的取出饼干”并不是一个真实的场景。
- 汤姆·波拉德和保罗A．吉安纳罗斯指出，在火车头案例中的某些数量有版本问题。
- 兰姆·林布发现了一个拼写错误，还建议了澄清章节。
- 2013春天在我的《贝叶斯统计计算》课程上，学生们提出了许多有益的修正和建议，他们是：凯·奥斯汀，克莱尔·巴尼斯，卡里·本德尔，瑞秋·铂伊，凯特·门多萨，阿琼·伊耶，本·克罗普，内森·林之，凯尔·麦克康诺亥，亚历克·雷德福，布伦丹·里特，埃文·辛普森。
- 格雷戈·马拉和马特·艾提帮我澄清了“正确的价格”这个问题的一些讨论。
- 马库斯·奥格伦指出火车头问题的原有声明是有些含糊的。
- O’Reilly Media的贾斯敏和丹在校对书的过程中也发现了许多可改进的地方。

第1章 贝叶斯定理

1.1 条件概率

所有贝叶斯统计的方法都基于贝叶斯定理，如果有条件概率的学习基础，意识到这一点很自然。因此我们会从概率、条件概率开始，然后到贝叶斯定理，最后讨论贝叶斯统计的内容。

概率表示为0和1之间的数字（包括0和1），含义是某一事件或者预测行为的可信程度，1值表示“事件为真”的情形肯定发生，或表述为预测成真；而0值则表示“事件为真”这一情形为假。

其他中间值表示确定性的程度。例如，0.5通常也会写成50%，意味着一个预测结果发生和不发生有同等可能性。例如，在一个掷硬币事件中，人像面（正面）朝上的概率就非常接近50%。

条件概率是带有某些（前提条件）背景约束下的概率问题。例如，我想了解一下明年自己心脏病发作的可能性。根据疾病控制中心的数据，每年大约有78.5万名美国人罹患心脏病（*http://www.cdc.gov//heartdisease/fact.html*）。

美国约有3.11亿人，假设随机挑选一个美国人，那么其在明年心脏病发作的概率大约是0.3%。

但就具体个例而言，“我”可不是那个被随意选中的美国人。流行病学家们已经明确了多种影响心脏病发作的风险因素，根据这些因素我的风险则有可能高于或低于平均值。

本人男，45 岁，有临界高胆固醇，这些因素增加了我发病的可能性；然而，血压低、不抽烟这些因素则降低了可能性。

把上面这些条件输入在线计算器*http://hp2010.nhlbihin.net/atpiii/calculator.asp*，我发现自己明年心脏病发作的风险约为0.2%，低于全国平均水平。这个值就是一个条件概率，因为它是基于一系列前提因素的，这些因素构成了我患心脏病的“条件”。

通常条件概率的记号是p(A |B)，表示在给定B 条件下A 事件发生的概率。在这个例子中，A 表示我明年罹患心脏病带的概率，而B 表示了上面所罗列的条件。

1.2　联合概率

联合概率：是指两个事件同时发生的概率。p（A 和B ）是A 和B 事件的发生都为真的概率。

如果你已经理解了投骰例子和它的背景，我们开始学习下面的公式：

p(A 和B)= p(A)p(B)　　提醒：表达式并非总是成立。

例如，如果我投掷两个硬币，A 表示第一枚硬币正面朝上，B 表示第二枚硬币正面朝上，那么p(A)= p(B) = 0.5，同样的p(A 和B) = p(A)p(B) = 0.25。

但是上面公式仅在A 和B 都是独立事件的情况下才成立。即：已知A 事件的结果并不影响或改变B 事件发生的概率。或更正式表示为，p(B |A)= p(B)。

再考虑另一个事件之间并不独立的例子。假设 A 表示今天下雨的事件，B 表示明天会下雨的事件。如果我已经知道今天下雨，则明天还有可能下雨（译注：与仅仅单独考虑某一天会下雨的概率相比较），所以p(B |A)＞p(B)。

通常意义下，联合概率表述为

$$p(A \text{ and } B) = p(A)\, p(B|A)$$

对于任何A 、B 事件，如果任意一天下雨的机会是0.5，连续两天就不会是0.25，而是可能更高一点。

1.3　曲奇饼问题

我们即将开始讨论到贝叶斯定理，但我还想通过一个被称为“曲奇

饼问题”的例子来介绍它。假设有两碗曲奇饼，碗1包含30个香草曲奇饼和10个巧克力曲奇饼，碗2有上述两种饼干各20个。

现在设想你在不看的情况下随机地挑一个碗拿一块饼，得到了一块香草曲奇饼。我们的问题是：从碗1取到香草曲奇饼的概率是多少？

这就是一个条件概率问题；我们希望得到概率p(碗1|香草)，但怎样进行计算并非显而易见。问题如果换成在碗1中香草曲奇饼的概率则简单得多。

$$p(香草|碗1)= 3/4$$

不巧的是，$p(A|B)$并不和$p(B|A)$相同，但有方法从一个计算出另一个：贝叶斯定理。

1.4 贝叶斯定理

现在，我们准备好进行贝叶斯定理推导需要的所有条件了。首先，我们注意到，联合概率是乘积可交换（乘法交换律）的，即：

$$p(A \text{ and } B) = p(B \text{ and } A)$$

对于任何A，B表示的事件都成立。

然后，我们写出一个联合概率的表达式：

$$p(A \text{ and } B) = p(A)p(B|A)$$

由于我们并没有明确定义A和B的含义，因而可以对A、B进行互换操作。

交换它们的位置：

$$p(B \text{ and } A) = p(B)p(A|B)$$

把这些表达式连接起来，我们得到下面的表达式：

$$p(B)p(A|B)= p(A)p(B|A)$$

这意味着我们有两种方式计算联合概率，已知p(A)，乘以p(B|A)；或者从另一方向，已知p(B)，乘以p(A|B)。两种方法是相同的。

最后，将上式除以p(B)，得到：

$$\mathrm{p}(A|B) = \frac{\mathrm{p}(A)\mathrm{p}(B|A)}{\mathrm{p}(B)}$$

这正是贝叶斯定理！看起来不起眼，不过它会显示出令人吃惊的强大之处。

例如，我们可以用它来解决曲奇饼问题。

假设B_1表示曲奇饼属于碗1的概率，V表示曲奇饼是香草曲奇饼的概率。

带入贝叶斯定理我们得到：

$$\mathrm{p}(B_1|V) = \frac{\mathrm{p}(B_1)\mathrm{p}(V|B_1)}{\mathrm{p}(V)}$$

等式左边就是我们希望得到的，一块香草曲奇饼来自碗1的概率。

等式的右边表示：

- p(B_1)：这是我们忽略得到曲奇饼这个条件时（零条件下）选中碗1的概率。因为选择碗的过程是随机的，我们可以假设p(B_1)=1/2。
- p(V|B_1)：这是从碗1得到一个香草曲奇饼的概率＝3/4。
- p(V)：从任意碗里得到一个香草曲奇饼的概率。因为考虑到选择碗的机会均等，而且每个碗的曲奇饼数量都是40，得到曲奇饼的机会是相同的。两个碗中香草和巧克力曲奇饼总数各是50和 30，因此p(V)＝5/8。

把它们放在一起，我们得到：

$$\mathrm{p}(B_1|V) = \frac{(1/2)(3/4)}{5/8}$$

结果是3/5。所以，“得到一块香草曲奇饼”是支持于假设“来自碗1”的证据，因为香草曲奇饼来自碗1的可能性更大。

这个例子演示了一个应用贝叶斯定理的案例：它提供了一个从p(*B* |*A*) 得到p(*A* |*B*)的策略。

这种策略在解决类似“曲奇饼问题”的情况下是有用的，即从贝叶斯等式的右边计算要比左边容易的情况下。

1.5　历时诠释

还有另外一种理解贝叶斯定理的思路：它给我们提供的是一种根据数据集*D* 的内容变化更新假设概率*H* 的方法。

这种对贝叶斯定理的理解被称为“历时诠释”。

“历时”意味着某些事情随着时间而发生；在本例，即是假设的概率随着看到的新数据而变化。

在考虑*H* 和*D* 的情况下，贝叶斯定理的表达式可以改写成：

$$\mathrm{p}(H|D) = \frac{\mathrm{p}(H)\mathrm{p}(D|H)}{\mathrm{p}(D)}$$

在这种解释里，每项意义如下：

- p(*H*)称为先验概率，即在得到新数据前某一假设的概率。
- p(*H* |*D*)称为后验概率，即在看到新数据后，我们要计算的该假设的概率。
- p(*D* |*H*)是该假设下得到这一数据的概率，称为似然度。
- p(*D*)是在任何假设下得到这一数据的概率，称为标准化常量。

有些情况，我们可以基于现有背景信息进行计算。比如在曲奇饼问题中，我们就将随机选中碗1或碗2的概率假设为均等。

在其他情况下，先验概率是偏主观性的；对某一先验概率，理性派的人可能会有不同意见，或许由于他们使用不同的背景信息做出判断，或者因为他们针对相同的前提条件做出了不同的解读。

似然度是贝叶斯计算中最简单的部分，在曲奇饼问题中曲奇饼来自来自哪个碗，则我们就计算那个碗中香草曲奇饼的概率。

标准化常量则有些棘手，它被定义为在所有的假设条件下这一数据出现的概率，但因为考虑的正是最一般的情况，所以不容易确定这个常量在具体应用场合的现实意义。

最常见的，我们可以指定一组如下的假设集来简化。

互斥的：集合中，至多一个假设为真。

完备的：集合中，至少一个假设必为真，且集合包含了所有的假设。

我使用suite这个词来表示具备上述属性的假设集。

在曲奇饼问题中，仅有两个假设：饼干来自碗1或者碗2，它们就是互斥的和完备的。

在本例中，我们可以用全概率公式计算p(D)，即如果发生某一事件有互不容的两个可能性，可以像下面这样累加概率：

$$\mathrm{p}(D) = \mathrm{p}(B_1)\mathrm{p}(D|B_1) + \mathrm{p}(B_2)\mathrm{p}(D|B_2)$$

代入饼干问题中的实际值，得到：

$$\mathrm{p}(D) = (1/2)(3/4) + (1/2)(1/2) = 5/8$$

我们早前心算得到的结果也是一样的。

1.6　M&M豆问题

M&M豆是有各种颜色的糖果巧克力豆。制造M&M豆的Mars公司会不时变更不同颜色巧克力豆之间的混合比例。

1995年，他们推出了蓝色的M&M豆。在此前一袋普通的M＆M豆中，颜色的搭配为：30%褐色，20%黄色，20%红色，10%绿色，10%橙色，10%黄褐色。这之后变成了：24%蓝色，20%绿色，16%橙色，14%

黄色，13%红色，13%褐色。

假设我的一个朋友有两袋M&M豆，他告诉我一袋是1994年，一袋是1996年。

但他没告诉我具体哪个袋子是哪一年的，他从每个袋子里各取了一个M&M豆给我。一个是黄色，一个是绿色的。那么黄色豆来自1994年的袋子的概率是多少？

这个问题类似于曲奇饼问题，只是变化了我抽取样品的方式（碗还是袋）。这个问题也给了我一个机会演示纸面方法：也就在仅仅在纸上画画就可以解决类似这样的问题（译注：作者为后续章节的计算型方法铺垫）。在下一章中，我们将以计算方法解这些问题。

第一步是枚举所有假设。取出黄色M＆M豆的袋子称为袋1，另一个称为袋2，所以假设是：

- A：袋1是1994年的，袋2是1996年的。
- B：袋1是1996年的，袋2是1994年的。

接着我们设计一个表格，每行表示每个假设，每列表示贝叶斯定理中的每一项：

先验概率**p(*H*)**		似然度**p(*D* $\mid$ *H*)**	**p(*H*) p(*D* $\mid$ *H*)**	后验概率**p(*H* $\mid$ *D*)**
假设A	1/2	(20) (20)	200	20/27
假设B	1/2	(10) (14)	70	7/27

第一列表示先验。基于问题的声明，选择p(*A*)=p(*B*)= 1/2是合理的。

第二列表示似然度，表明了问题的背景信息。举例来说，如果A为真，黄色M＆M是来自1994年的袋概率20%，而绿色来自1996包的概率为20%。因为选择是独立的，我们将其相乘以得到联合概率。

第三列由前两列得到。此列的总和270是归一化常数（译注：参考全概率公式）。为了得到最后一列的后验概率，我们将第三列的值归一化后得到第四列的值。

就是这样。简单吧？

还有，你可能会被一个细节所困扰。我将p(D |H)写成了百分数的形式而不是概率形式，这意味着它没有除以因子10000。但是当我们将其除以归一化常数时就抵消了，因此这不影响结果。

当设定的假设是互斥和穷举的，你可以将似然度乘以任何因子，如果方便，将同一个因子应用到整列上。

1.7 Monty Hall难题

蒙蒂大厅（Monty Hall problem）难题可能是历史上最有争议的概率问题。问题看似简单，但正确答案如此有悖常理以致很多人不能接受，很多聪明人都难堪于自己搞错了反而据理力争，而且是公开的。

蒙蒂大厅是游戏节目“来做个交易”（Let’s Make a Deal）的主场。蒙蒂大厅难题也是这一节目的常规游戏之一。如果你参加节目，规则是这样的：

- 蒙蒂向你示意三个关闭的大门，然后告诉你每个门后都有一个奖品：一个奖品是一辆车，另外两个是像花生酱和假指甲这样不值钱的奖品。奖品随机配置。
- 游戏的目的是要猜哪个门后有车。如果你猜对了就可以拿走汽车。
- 你先挑选一扇门，我们姑且称之为门A，其他两个称为门B和门C。
- 在打开你选中的门前，为了增加悬念，蒙蒂会先打开B或C中一个没有车的门来增加悬念（如果汽车实际上就是在A门背后，那么蒙蒂打开门B或门C都是安全的，所以他可以随意选择一个）。
- 然后蒙蒂给你一个选择。坚持最初的选择还是换到剩下的未打开的门上。

问题是，你应该“坚持”还是“换”？有没有区别？

大多数人都有强烈的直觉，认为这没有区别。剩下两个门没有打开，车在门A背后的机会是50%。

但是，这是错的。事实上，如果你坚持选择门A，中奖概率只有1/3；而如果换到另外一个门，你的机会将是2/3。

运用贝叶斯定理，我们可以将这个问题分解成几个简单部分，也许这样可以说服自身，“正确”的答案实际上的的确确是对的。

首先，我们应该对数据进行仔细描述。在本例中为*D* 包括两个部分：蒙蒂打开了门B，而且没有车在后面。

接下来，我们定义了三个假设：A，B和C，表示假设车在门A，门B，或门C后面。同样，采用表格法：

先验概率**p(*H*)**		似然度**p(*D* |*H*)**	**p(*H*) p(*D* |*H*)**	后验概率**p(*H* |*D*)**
假设A	1/3	1/2	1/6	1/3
假设B	1/3	0	0	0
假设C	1/3	1	1/3	2/3

填写先验很容易，因为我们被告知奖品是随机配置的，这表明该车可能在任何门后面。

定义似然度需要一些思考，在充分合理的考虑后，我们确信正确的似然度如下：

- 考虑假设A：如果汽车实际上是在门A后，蒙蒂可以安全地打开门B或门C。所以他选择门B的概率为1/2。因为车实际上是在门A后，也就是说车不在门B后的概率是1。
- 考虑假设B：如果汽车实际上是在门B后，蒙蒂不得不打开门C，这样他打开门B的概率就是0（译注:也就是这个假设的似然度为0，不可能发生）。
- 最后考虑假设C：如果车是在门C后，蒙蒂打开门B的概率为1，发现车不在那儿的概率为1（译注：因为在选手已经选了A门这个情况下，可供蒙蒂增加悬念开门的选择只有B和C，而假设C有车，蒙蒂肯定不会选，因此蒙蒂会打开B门的概率为1，也就是在这个假设下，数据D的似然度为1）。

现在我们已经完成有难度的部分了，剩下无非就是算术。第三列的总和为1/2，除以后得到p(*A* |*D*) = 1/3，p(*C* |*D*) = 2/3，所以你最好是换个选择。

该问题有许多变形。贝叶斯方法的优势之一就是可以推广到这些变

形问题的处理上。

例如，设想蒙蒂总是尽可能选择门B，且只有在迫不得已的时候才选门C（比如车在门B后）。在这种情况下，修正后的表如下：

先验概率**p(*H*)**		似然度**p(*D* |*H*)**	**p(*H*) p(*D* |*H*)**	后验概率**p(*H* |*D*)**
假设A	1/3	1	1/3	1/2
假设B	1/3	0	0	0
假设C	1/3	1	1/3	1/2

唯一的变化是p(*D* |*A*)。如果车在门A后，蒙蒂可以选择打开B或C。但在这个变形问题里面，他总是选择B，因此p(*D* |*A*) = 1。

因此，对A和C，似然度是相同的，后验也是相同的：p(*A* |*D*) = p(*C* |*D*) = 1/2，在这种情况下，蒙特选择B 门显示不了车位置的任何信息，所以无论选手选择坚持不变还是改变都无关紧要。

反过来的情况下，如果蒙蒂打开门C，我们就知道p(*B* |*D*) = 1（译注：因为蒙蒂总是优先选择门B，而门D是他打开了门C，因此在假设车在门B后的前提下，他必然会打开门C，概率为1，即p(*B* |*D*)=1）。

本章中我介绍了蒙蒂问题，因为我觉得这里有它的趣味性，也因为贝叶斯定理使问题的复杂性更易控制。但这并不算是一个典型的贝叶斯定理应用，所以如果你觉得它令人困惑，没什么好担心的！

1.8 讨论

对于涉及条件概率的很多问题，贝叶斯定理提供了一个分而治之的策略。如果p(*A* |*B*)难以计算，或难以用实验衡量，可以检查计算贝叶斯定理中的其他项是否更容易，如p(*B* |*A*)，p(*A*)和p(*B*)。

如果蒙蒂大厅问题让你觉得有趣，我在一篇文章“All your Bayes are belong to us”中收集了很多类似问题，你可以去单击链接进行阅读 *http://allendowney.blogspot.com/2011/10/all-your-bayes-are-belong-to-us.html* 。

第2章　统计计算

2.1　分布

在统计上，分布 是一组值及其对应的概率。

例如，如果滚动一个六面骰子，可能的值是数字1至6，与每个值关联的概率是1/6。

再举一个例子，你应该有兴趣了解在日常的英语使用中每个单词出现的次数。你可以建立一个包含每个字及它出现的次数的分布。

为了表示Python中的分布，可以使用一个字典映射某个值和它的概率。我编写了一个名为`Pmf` 的类，利用Python字典实现了上述功能，而且提供了一些有用的方法。为了对应概率质量函数 这种分布的数学表示法，我将其命名为Pmf。

`Pmf` 的定义在一个我为本书完成的Python模块`thinkbayes.py` 中。可以从http://thinkbayes.com/thinkbayes.py 下载。欲了解更多信息参见前言的“代码指南”。

要使用`Pmf`，可如下导入：

```
from thinkbayes.py  import  Pmf
```

下面的代码建立一个Pmf来表示六面骰子的结果分布：

```
pmf = Pmf()
for x in [1,2,3,4,5,6]:
    pmf.Set(x, 1/6.0)
```

`Pmf` 创建一个空的没有赋值的pmf。`Set` 方法设置每个值的概率为1/6。

这里是另一个例子，计算每个单词在一个词序列中出现的次数：

```
pmf = Pmf()
for word in word_list:
    pmf.Incr(word, 1)
```

`Incr` 为每个单词的相应“概率”加1。如果一个词还没有出现在Pmf中，那么就将这个词添加进去。

我把“概率”加上引号是因为在这个例子中概率还没有归一化，也就是说它们的累加和不是1，因此不是真正的概率。但在本例中单词计数与概率成正比。所以当完成了所有的计数，就可以通过除以计数的总值来计算得到概率。

`Pmf` 提供了一种`Normalize` 方法来实现上述功能：

```
pmf.Normalize()
```

一旦有一个Pmf对象，你可以像下面这样得到任何一个值相关联的概率：

```
print pmf.Prob('the')
```

这会打印输出单词“the”在词序列中出现的频率。

Pmf使用Python字典来存储值及其概率，所以Pmf中的值可以是任意可被哈希的类型。概率可以是任意数值类型，但通常是浮点数（`float` 类型）。

2.2 曲奇饼问题

在贝叶斯定理的语境下，可以很自然地使用一个Pmf映射每个假设和对应的概率。

在曲奇饼问题里面，该假设是B_1 和B_2 。在Python中可以使用字符

串来表示它们：

```
pmf = Pmf()
pmf.Set('Bowl1', 0.5)
pmf.Set('Bowl2', 0.5)
```

这一分布包含了对每个假设的先验概率，称为先验分布 。

要更新基于新数据（拿到一块香草曲奇饼）后的分布，我们将先验分别乘以对应的似然度。

从碗1拿到香草曲奇饼的可能性是3/4，碗2的可能性是1/2。

```
pmf.Mult('Bowl1', 0.75)
pmf.Mult('Bowl2', 0.5)
```

如你所愿，`Mult` 将给定假设的概率乘以已知的似然度。

更新后的分布还没有归一化，但由于这些假设互斥且构成了完全集合（意味着完全包含了所有可能假设），我们可以进行重新归一化 如下：

```
pmf.Normalize()
```

其结果是一个包含每个假设的后验概率分布，这就是所说的后验分布 。

最后，我们可以得到假设碗1的后验概率如下：

```
print pmf.Prob('Bowl 1')
```

答案是0.6。你可以从 *http://thinkbayes.com/cookie.py* 下载这个例子。欲了解更多信息，请参见前言的“代码指南”。

2.3 贝叶斯框架

在继续讨论其他的问题前，我想在上一节的基础上重写代码以使其更通用。首先我将定义一个类来封装与此相关的代码：

```
class Cookie(Pmf):

    def __init__(self,hypos):
        Pmf.__init__(self)
        for hypo in hypos:
            self.Set(hypo,1)
        self.Normalize()
```

Cookie对象是一个映射假设到概率的Pmf对象。`__init__` 方法给每个假设赋予相同的先验概率。如上一节中就有两种假设：

```
    hypos= ['Bowl1', 'Bowl2']
    pmf =Cookie(hypos)
```

`Cookie` 类提供了`Update` 方法，它以data为参数并修正相应的概率：

```
    def Update (self,data):
        for hypo in self.Values():
            like= self.Likelihood(data,hypo)
            self.Mult(hypo, like)
        self.Normalize()
```

`Update` 遍历suite中的每个假设，并将其概率乘以数据在某一假设下的似然度，似然度由`Likelihood` 计算：

```
   mixes = {
       'Bowl 1':dict(vanilla=0.75, chocolate=0.25),
       'Bowl 2':dict(vanilla=0.5, chocolate=0.5),
       }

   def Likelihood(self, data, hypo):
       mix = self.mixes[hypo]
```

```
        like = mix[data]
        return like
```

Likelihood 使用mixes ， 它使用Python的字典结构来映射碗名和在碗中曲奇饼的混合比例。

如下面这样进行更新：

```
    pmf.Update('vanilla')
```

然后我们就可以打印输出每个假设的后验概率：

```
    for hypo , prob in pmf.Items():
        print  hypo, prob
```

其结果是

```
Bowl 1  0.6
Bowl 2  0.4
```

结果和我们之前得到的结论一样。比起我们在前面章节看到的，这段代码更复杂。

一个优点是，它可以推广到从同一个碗取不只一个曲奇饼（带替换）的情形：

```
    dateset= ['vanilla', 'chocolate', 'vanilla']
    for data in dataset:
        pmf.Update(data)
```

另一优点是，它提供了解决许多类似问题的框架。在下一节中，我们将解决蒙蒂大厅问题的计算，然后看看框架的哪些部分相同。

本节中的代码可以从 *http://thinkbayes.com/cookie2.py* 获得。欲了解更多信息，请参见前言的“代码指南”。

2.4 Monty Hall难题

为了解决蒙蒂大厅（Monty Hall）问题，我将定义一个新的类：

```
class monty(Pmf):

    def __init__(self, hypos):
        Pmf.__init__(self)
        for hypo in hypos:
            self.Set(hypo, 1)
        self.Normalize()
```

到目前为止，蒙蒂大厅和曲奇饼是完全一样的。创建Pmf的代码也一样，除了假设的名称：

```
    hypos='ABC'
    pmf =Monty(hypos)
```

对Update的调用几乎是相同的：

```
    data='B'
    pmf.Update(data)
```

Update的实现也是完全一样的：

```
    def Update (self, data):
        for hypo in self.Values ():
            like = self.Likelihood(data, hypo)
            self.Mult(hypo,like)
        self.Normalize()
```

唯一需要些额外工作的是Likelihood：

```
    def Likelihood (self,data,hypo):
        if hypo==data:
            return 0
        elif hypo=='A':
```

```
        return 0.5
    else:
        return 1
```

最后，打印输出的结果是一样的：

```
for hypo, prob in pmf.Items():
    print hypo,prob
```

答案是

```
A 0.333333333333
B 0.0
Ç 0.666666666667
```

在本例中，Likelihood 的编写有一点点复杂，但该贝叶斯框架的 Update很简单。本节中的代码可以从 *http://thinkbayes.com/monty.py* 获得。欲了解更多信息，请参见前言的“代码指南”。

2.5 封装框架

现在，我们看看框架的哪些元素是相同的，这样我们就可以把它们封装进一个Suite 对象，即一个提供__init__，Update 和Print 方法的pmf 对象：

```
class Suite(Pmf)
    “代表一套假设及其概率。”

    def __init__(self, hypo=tuple()):
        “初始化分配。”

    def Update(self, data):
        “更新基于该数据的每个假设。”

    def Print (self):
        “打印出假设和它们的概率。”
```

Suite 的实现在thinkbayes.py 中。要使用Suite 对象，你应当编写一个继承自Suite 的类，并自行提供Likelihood 方法的实现。例如，这是一个以蒙蒂大厅问题改写的使用Suite 的方案：

```
from  thinkbayes  import Suite
class Monty(Suite):
    def Likelihood (self,data,hypo):
        if hypo ==data:
            return 0
        elif hypo=='A':
            return 0.5
        else:
            return 1
```

而下面是一个使用这个类的代码：

```
    suite=Monty('ABC')
    suite.Update('B')
    suite.Print()
```

你可以从 *http://thinkbayes.com/monty2.py* 下载这个例子。更多信息见前言的“代码指南”。

2.6　M&M豆问题

我们可以使用Suite 框架来解决M&M豆的问题。除了编写Likelihood 有点棘手，其他一切都很简单。

首先，需要对1995年之前和之后的颜色混合情况进行封装：

```
mix94=dict(brown= 30,
           yellow= 20,
           red= 20,
           green= 10,
           orange= 10,
           tan= 10)

    mix96=dict(blue= 24,
               green= 20,
```

```
                 orange= 16，
                 yellow= 14，
                 red= 13，
                 brown= 13)
```

然后，封装假设：

```
hypoA =dict(bag1 = mix94，bag2 = mix96)
hypoB =dict(bag1 = mix96，bag2 = mix94)
```

hypoA 表示假设袋1是1994年，袋2是1996年。hypoB 是相反的组合。

接下来，从假设的名称来映射其含义：

```
hypotheses=dict(A=hypoA，B=hypoB)
```

最后，开始编写Likelihood 。在这种情况下，假设hypo 是一个A或B 的字符串，数据是一个指定了袋子年份和颜色的元组。

```
def Likelihood(self, data, hypo):
    bag, color = data
    mix = self.hypotheses[hypo][bag]
    like = mix[color]
    return like
```

下面是创建该suite对象并进行更新的代码：

```
suite = M_and_M('AB')

suite.Update(('bag1', 'yellow'))
suite.Update(('bag2', 'green'))

suite.Print()
```

结果如下：

```
A 0.740740740741
B 0.259259259259
```

A的后验概率大约是20/27，正是我们之前得到的。

本节中的代码可以从 *http://thinkbayes.com/m_and_m.py* 获得。欲了解更多信息，请参见前言的“代码指南”。

2.7 讨论

本章介绍了Suite类，它封装了贝叶斯update框架。

`Suite` 是一个抽象类（abstract type），这意味着它定义了`Suite` 应该有的接口，但并不提供完整的实现。`Suite` 接口包括`Update` 和`Likelihood` 方法，但只提供了`Update` 的实现，而没有`Likelihood` 的实现。

具体类（concrerte type）是继承自抽象父类的类，而且提供了缺失方法的实现。例如，`Monty` 扩展自`Suite` ，所以它继承了`Update` 并且实现了`Likelihood` 。

如果你熟悉设计模式，你可能会意识到这其实是设计模式中模板方法的一个例子。你可以从 *http://en.wikipedia.org/wiki/Template_method_pattern* 了解这个模式。

大多数在以下章节中的例子遵循相同的模式，对于每个问题我们定义一个扩展的`Suite` 对象，继承了`Update` 方法，并提供了`Likelihood` 。在少数情况下，会重写`Update` 方法，通常是为了提高性能的缘故。

2.8 练习

练习2-1。

在第11页的“贝叶斯框架”里面，我提到了曲奇饼问题的解法是简化的，是曲奇饼有补充的取多个饼的情况（有放回的情况）。

但更可能的情况是，我们吃掉了取出的曲奇饼，那么似然度就依赖于之前的取曲奇饼行为（曲奇饼少了）。

修改本章中的解法以处理没有曲奇饼补充的情况。提示：添加Cookie 的实例变量来表示碗的假设状态，并据此修改似然度。你可能要定义一个Bowl 对象。

第3章　估计

3.1　骰子问题

假设我有一盒骰子，里面有4面的骰子、6面的骰子、8面的骰子、12面的骰子和20面的骰子各1个。如果曾经玩过游戏《龙与地下城》，你当然会明白我的所指。

假如我随机从盒子中选一个骰子，转动它得到了6 。那么每一个骰子被选中的概率是多少？

我通过一个三步策略来解决这个问题。

1．选择假设的表示方法。

2．选择数据的表示方法。

3．编写似然度函数。

在前面的例子中我用字符串来表示假设和数据，但骰子问题中我将使用数字。

确切地说我将使用整数4、6、8、12和20来表示假设：

```
suite=Dice([ 4，6，8，12，20 ])
```

另外，从1到20的整数作为数据。有了这些表达式，编写似然函数就很简单了：

```
class Dice(Suite):
    def Likelihood(self, data, hypo):
        if hypo < data:
            return 0
        else:
            return 1.0/hypo
```

这里Likelihood的原理是：如果hypo<data，意味着投骰子值大于骰子的面数，显然这是一个不可能的情形，所以似然度是0。

另外的情形下，问题变成“考虑到所有假设的点数，得到某个点数结果的机会是多少？”

答案是1/hypo无论数据是什么。

下面是使用update（如果转动得到6）的语句声明：

```
suite.Update (6)
```

后验分布的结果如下：

```
4 0.0
6 0.392156862745
8 0.294117647059
12 0.196078431373
20 0.117647058824
```

当得到6后，骰子是4面的概率是0。最可能的备选是6面骰，但也有约12%的可能是20面骰。

如果我们多摇几次，得到6，8，7，7，5，4这样一组数据的情况下呢？

```
    for roll in [ 6，8，7，7，5，4 ] :
        suite.Update (roll)
```

结合该组数据，可以去掉6面骰的可能性了（因为有大于6的值），8面骰看起来可能性很大。

结果如下：

```
4 0.0
6 0.0
8 0.943248453672
```

```
12 0.0552061280613
20 0.0015454182665
```

现在有94%的可能我们转动了一个8面骰，同时还有1%可能是一个20面骰。

骰子问题参考了我在Sanjoy Mahajan的贝叶斯推论课上看到的一个例子。

你可以从 *http://thinkbayes.com/dice.py* 下载代码。欲了解更多信息，请参见前言的“代码指南”。

3.2 火车头问题

我是在弗雷德里克·莫斯泰勒的《五十个概率难题的解法》（多佛出版社，1987）一书中发现火车头问题的：

> 铁路上以1到N命名火车头。有一天你看到一个标号60的火车头，请估计铁路上有多少火车头？

基于这一观察结果，我们知道铁路上有60个或更多的火车头。但这个数字到底是多少？

要应用贝叶斯进行推理，我们可以将这个问题分成两步骤进行：

1．在得到数据之前，我们对N的认识是什么？

2．已知一个N的任意值后，得到数据（“标志为60号的火车头”）的似然度？

第一个问题的答案就是问题的前置概率。第二个问题是似然度。

在选择前置概率上，我们还没有太多的基本信息，但我们可以从一些简单情况开始，再考虑进一步的方法。假设N可以是从1到1000等概率的任何值。

```
    hypos= xrange (1，1001)
```

接着我们需要的是一个似然函数。先假设存在一个有N个火车头的车队，我们看到60号火车头的概率是多少？假设只有一个列车运营公司（或者只有一个我们关注的公司），看到任意一个火车头有同等可能，那么看到的任何特定火车头的机会为$1/N$。

似然度函数如下：

```
class Train(Suite) :
    def Likelihood(self，data，hypo) :
        if hypo<data：
            return 0
        else：
            return 1.0/hypo
```

看起来很熟悉，似然函数在火车头问题和骰子问题上是相同的。

Update 如下：

```
    suite=Train (hypos)
    suite.Update (60)
```

因为有太多的假设（1000）要打印输出，所以我绘制了如图3-1所示的结果。意料之中的是，N中60以下的所有值都被去掉了。

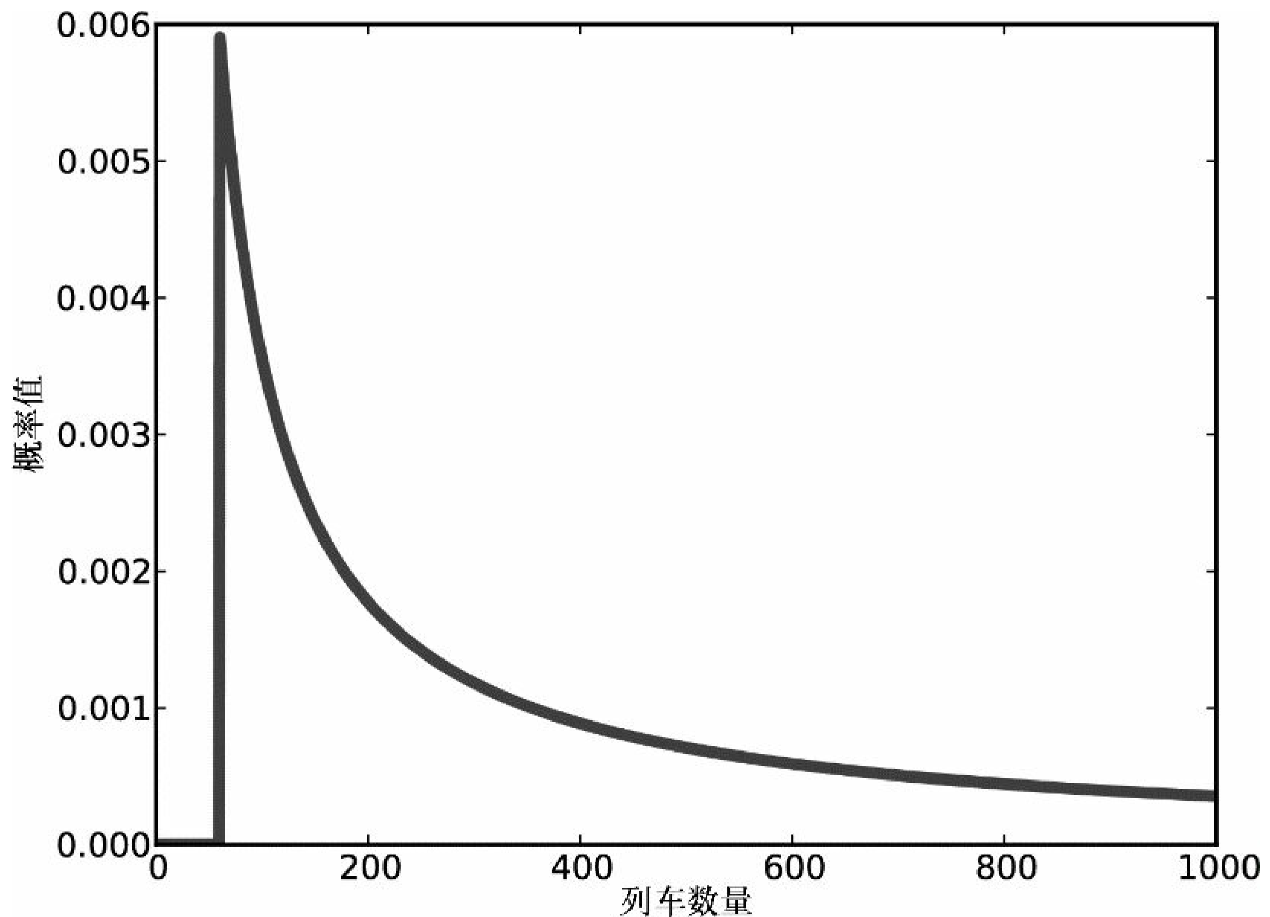


图3-1　火车头问题的后验分布，基于均匀分布的先验

如果非要猜测的话，最可能的值是60。这似乎算不上很好的结果，毕竟，想想你恰好看到最高标志号火车头的机会是多少呢（应该不高吧）？不过，如果想使猜到的答案完全正确的可能性最大化，你应该猜60。

不过，这还不是我们的目标。另一个可选的方法是计算后验概率的平均值分布：

```
def Mean(suite):
    total= 0
   for hypo, prob in suite.Items ():
        total + = hypo*prob
return total
print Mean(suite)
```

或者你可以用由Pmf提供的非常类似的方法：

```
print suite.Mean ()
```

后验的平均值是333，所以要是你想最大限度地减少错误，这也许是一个很好的猜测结果。

如果你一遍又一遍地玩这个猜谜游戏，使用后验概率的平均值来作为估计值会减少从长远来看的均方差（参考 *http://en.wikipedia.org/wiki/Minimum_mean_square_error* ）。

你可以从 *http://thinkbayes.com/train.py* 下载这个例子。如需更多信息参见前言的"代码指南"。

3.3 怎样看待先验概率?

为了进一步解决火车头问题，我们必须进行一些假设。

其中一些是相当武断的。尤其当我们选择了在1-1000间的均匀随机分布的先验概率。其实并没有特别的理由选择1000作为上界或均匀随机来作为分布特征。

相信一个运营着1000个火车头的铁路公司算不上疯狂，但一个理性点的人可能会对这一问题做出更多或更少的猜测。

因此，我们可能想知道是否后验概率分布对这些假设敏感。仅仅依赖一次观察的小量数据，情况可能真是如此的（敏感）。

回忆一下，从1到1000的均匀分布的先验概率，后验概率的平均值是333。同样上界为500，我们得到的后验平均值为207，一个2000的上界后验平均值为552。

所以结果很糟（猜测结果对上界敏感）。有两种方法继续进行分析：

- 获取更多的数据。
- 更多的背景信息。

有了更多的数据后，基于不同的先验概率，后验分布趋于收敛。

例如，假设除了列车60我们也看到列车30和90。我们可以这样更新分布：

```
for data in[60，30，90]:
    suite.Update (data)
```

采用这些数据时，后验概率的均值是

上限	后验均值
500	152
1000	164
2000	171

这样差异就较小了。

3.4 其他先验概率

如果没有更多的数据，另一个方法是通过收集背景资料优化先验。

大型和小型公司有同等可能性的假设也许相当不合理，大型公司可能有1000台火车头，小型公司仅有1台火车头。

通过一些努力，我们很有可能发现在观察区域内火车运营公司的清单，或者可以采访铁路运输专家来收集这些公司一般规模的信息。

但是，即使没有深入了解铁路产业的一些具体情况，我们也可以做一些猜测。在大多数领域，有大量小型公司，一些中型公司，一个到两个非常大型的公司。

事实上，公司规模的分布往往遵循幂律，参考罗伯特·阿克斯特尔在《科学》杂志上的报道（http://www.sciencemag.org/content/293/5536/1818.full.pdf）。

这规律表明，如果少于10个火车头的公司有1000家，100个火车头的公司可能有100家，1000个火车头的公司有10家，10000个火车头的公司可能仅有1家。

在数学上，幂律表示公司的数量与公司规模成反比，或

$$\mathrm{PMF}(x) \propto \left(\frac{1}{x}\right)^{\alpha}$$

其中PMF(x)是x的概率质量函数，α是一个通常接近于1的参数。

我们可以构造一个服从幂律分布的先验如下：

```
class Train (Dice):

    def__init__ (self, hypos, alpha = 1.0):
        Pmf.__init__ (self)
        for hypo in hypos:
            self.Set (hypo, hypo** (-alpha))
        self.Normalize ()
```

而下面是生成前置概率的代码：

```
    hypos=range(1,1001)
    suite=Train(hypos)
```

再次说明，上限是任意的，但对于幂律分布的先验概率，后验概率对这一选择的敏感性较小。

图3-2表示了基于幂律后与之前基于均匀随机后验概率的比较。

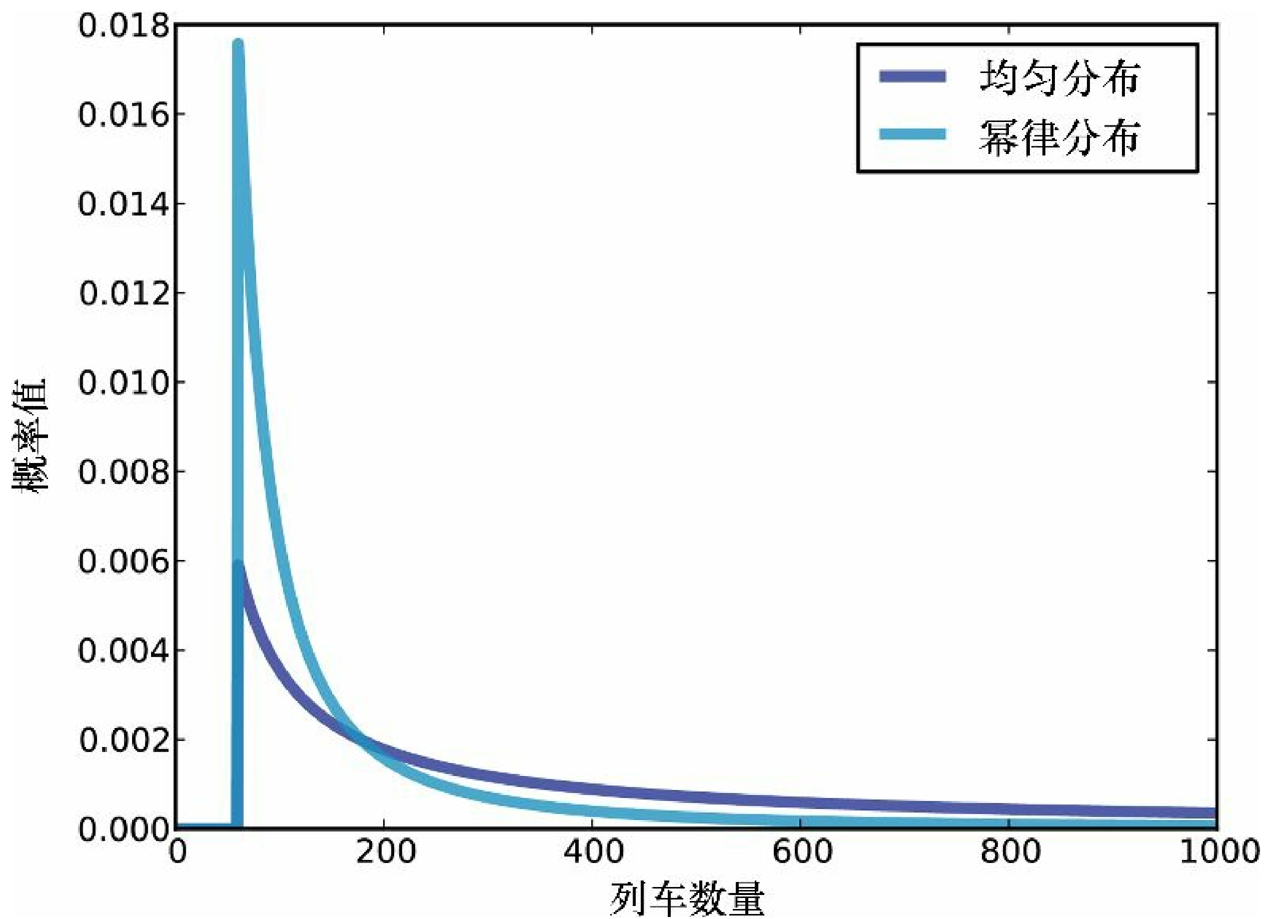


图3-2　前验为幂律的后验分布，对比于均匀前验分布

考虑了幂律分布作为背景资料后。我们可以消除大于700的N值了。

如果基于这种先验概率，在观察到列车30、60和90时，后验概率分别是：

上限	后验均值
500	131
1000	133
2000	134

现在的差异小得多了（考虑不同上界的假设）。事实上，考虑一个任意大的上界，平均值都收敛于134。

所以基于幂律分布的先验概率是比较现实的，因为它基于公司规模的一般情况，并且在实际中表现得更好。

你可以从 *http://thinkbayes.com/train3.py* 下载本节程序的例子。欲了解更多信息，请参见前言的“代码指南”。

3.5 置信区间

一旦计算出的后验分布，通过单点估计或区间对后验分布进行总结通常是有用的。

对于点估计，通常使用平均数、中位数或最大似然值。

对于区间，我们通常给出两个计算值，使得未知量有90%的可能落入这两个值之间（或者任何其他的概率值）。这些值定义了一个置信区间 。

计算置信区间的一个简单方法是在后验概率分布中累加其中的概率，并记录对应于概率5%和95%的值。也就是说，第5和第95百分位。

`thinkbayes` 提供了一个函数计算百分位数：

```
def Percentile(pmf, percentage):
    p = percentage / 100.0
    total = 0
    for val, prob in pmf.Items():
        total += prob
        if total >= p:
            return val
```

下面是一个其应用代码：

```
    interval = Percentile(suite, 5), Percentile(suite, 95)
    print interval
```

在前面的示例（看到了三个火车，且呈幂律分布的先验概率的火车头问题）中90%置信区间为（91,243）。如此大的范围其实确切的表明，（尽管平均值收敛了）我们仍然相当不确定究竟有多少火车头存在。

3.6　累积分布函数

在上一节中，我们通过数值迭代计算出百分比和Pmf的概率。如果我们需要计算多个百分位数，更有效方法是使用累积分布函数，或Cdf。

由于包含有某个分布的相同的信息，在这个意义上Cdf和Pmf是等价的，并可以随时从一个转换到另一个。Cdf的优点是可以更有效地计算百分位数。

`thinkbayes` 有一个表示累积分布函数的`Cdf` 类。`Pmf` 提供了一种方法生成相应的Cdf：

```
cdf = suite.MakeCdf ()
```

`Cdf` 提供了一个名为`Percentile` 的函数

```
interval = cdf.Percentile(5), cdf.Percentile(95)
```

将Pmf转换为Cdf需要正比于值数量`len(pmf)` 的运算时间。Cdf将值和概率存储在有序列表里（list），所以查询某个概率得到相应的值需要“对数时间（log time）”：即，时间和值的数量的对数成正比。查询一个值获得对应的概率也是对数时间，所以Cdf对于很多计算来说都是有效的。

本节中的例子位于 *http://thinkbayes.com/train3.py* 。欲了解更多信息参见前言的“代码指南”。

3.7 德军坦克问题

第二次世界大战期间，在伦敦的美国大使馆战争经济部门使用统计分析来估计德国生产的坦克和其他装备[①]。

西方盟军获得了一份记录簿，记录了存货和修理记录，其中包括坦克的底盘和发动机的序列号。

这些记录的分析表明，制造商为坦克类型分配了以100为一个区间的序列号，每个区间内的号码都是成序列的，但并不是每个区间内的数字都用到了。如此一来，在每个100的区间内，估算德军坦克问题的范围就可以按照之前的火车头问题来缩小了。

基于这种认识，美国和英国的分析师们完成了远远小于源自情报部门其他形式情报的估算结果。而在战后，记录显示这些结果实质上更准确。

他们也对轮胎、卡车、火箭等设备进行类似的分析，产生准确实用的经济情报。

德军坦克问题是个历史上有趣的问题，也是一个很好的在现实世界应用统计估计的例子。到目前为止，本书的许多例子都是游戏性质的问题，但我们马上会开始开始解决实际问题。我认为它是贝叶斯分析的优点，特别是在我们正在使用的计算方法上，它提供了一条从基础介绍到研究前沿的捷径。

3.8 讨论

贝叶斯当中，有两种途径选择先验分布。一些人建议选择最能代表问题相关背景资料的先验概率，在这种情况下，先验被认为是“信息”。问题是，人们可能会使用不同的背景信息（或者进行不同的诠释）。所以基于信息的先验往往显得主观。

另一种方法是所谓的“无信息参考的先验”，其目的是为了让数据来说话，越没有约束越好。在某些情况下，你可以选择包含一些期望属性的特殊先验，例如，就估计量设置一个最小先验。

“无信息先验”观点存在是因为它们似乎更为客观。但通常，我倾向使用先验信息。为什么呢？首先，贝叶斯分析总是基于模型决策的。选择先验就是决策之一，但它不是唯一的部分，甚至可能不是最主观的。因此，即使无信息先验较为客观，整个分析本身仍然是主观的。

另外，对于大多数实际中的问题，你很可能是在两个（对立面）之间：也许有大量的数据，也许没有。如果你有大量的数据，先验的选择不是特别关键；信息先验和无信息先验会得到几乎相同的结果。我们会在下一章看到类似的例子。

不过，如果像火车头问题，如果你没有太多的参考数据，那么采用相关的背景信息(如幂律分布)就有很大区别了。

而比如德军坦克问题，如果必须基于你的结论做出生死存亡的决策，你就应该利用所有的信息，而不是在“要保持客观“的幻觉中假装不了解具体情况。

3.9 练习

练习3-1。

为火车头问题写一个似然函数，我们必须要回答这个问题：“如果铁路上有N个火车头，我们看到60号的概率是多少？”

答案取决于当我们观察到火车头时，所采用的取样过程。

在本章中，我通过指定只有一个列车运营公司（或只有一个我们关心）解决了这个模棱两可的问题。

但是假设有很多家使用不同火车头号码编排的公司，再假设看到任意公司经营的任意火车的可能性相同。在这种情况下，似然函数是不同的，因为你更有可能看到一列由大型公司运营的火车。

作为练习，实现火车头似然函数的这种变化，并比较结果。

---

① 拉格尔斯，布罗迪.实证方法经济情报大战[J].《美国统计协会》杂志，1947，卷42，第237号。

第4章　估计进阶

4.1　欧元问题

在《信息论：推理和学习算法》一书中，大卫·麦凯提出了这样一个问题：

> 2000年1月4日，星期五，《卫报》上刊载了一个统计相关的声明：
>
> 当以边缘转动比利时一欧元硬币250次后，得到的结果是正面140次反面110次。“这看起来很可疑”，伦敦经济学院的统计讲师巴里·布莱特说，“如果硬币是均匀的，得到这个结果的可能性低于7%。”
>
> 那么，这一结果是否对‘硬币偏心而非均匀’提供了证据呢？

我们将采取以下两个步骤回答这个问题。第一是估计该硬币正面朝上的概率。第二是评估该数据是否支持了硬币偏心的假设。

你可以从 *http://thinkbayes.com/euro.py* 下载代码。欲了解更多信息，请参见前言的“代码指南”。

以边缘旋转任一硬币时都有正面朝上的概率x。有理由相信x的值取决于硬币的一些物理特性，主要是重量的分布。

如果硬币完全均匀，我们预计x应该接近50%，但对于一个不匀称的硬币，x应该会大有差别。我们可以用贝叶斯定理和观测到的数据来估计x。

我们定义一个假设101，其中H_x是假设正面朝上的概率x%，其值从0到100。首先以均匀前置概率开始，其中H_x的概率对所有的x是相同的。后面我们再考虑其他先验概率的情况。

likelihood函数相对容易：如果H_x为真，正面向上的概率为$x/100$而反面

向上的概率为$1-x/100$。

```
class Euro(Suite):

    def likelihood (self, data, hypo):
        x =hypo
        if data== 'H':
            return x/100.0
        else:
            return 1 - x/100.0
```

下面是实现suite和update的代码：

```
    suite=Euro (xrange (0 ,  101))
    dataset = 'H' * 140 + 'T' * 110

    for data in dataset:
        suite.Update (data)
```

其结果如图4-1所示。

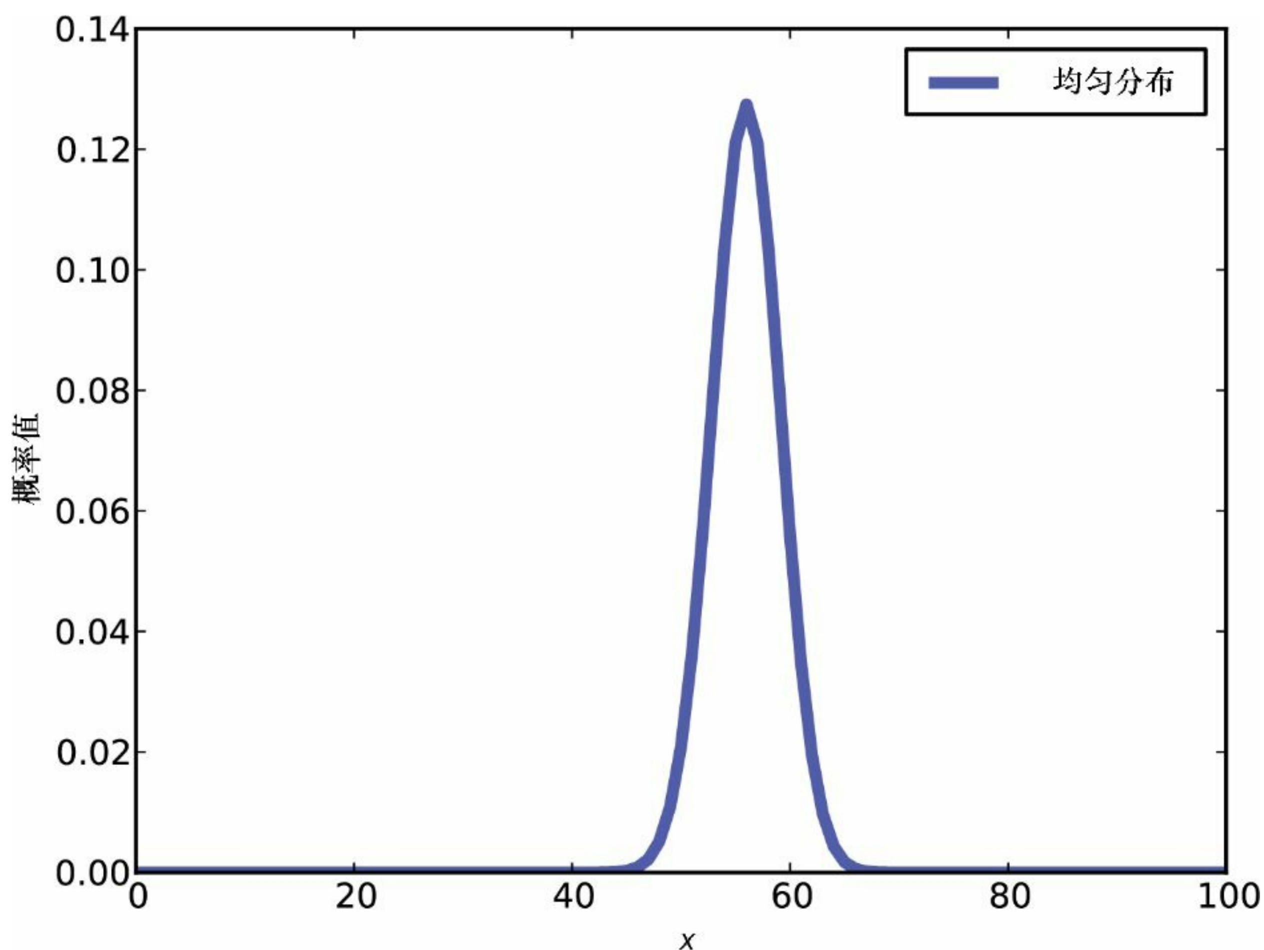


图4-1　前验为均匀分布的前提下，欧元问题的后验分布

4.2　后验概率的概述

总结一下，有几种方式来概括后验分布的特征。一种选择是找到后验分布的最大似然值。

thinkbayes 提供了一个函数实现：

```
def MaximumLikelihood(pmf) :
    “返回具有最高概率的值。”
    prob，val = max((prob，  val)for val,prob in  pmf.Items ())
    return val
```

在这种情况下，结果是56，这也是观察得到的百分比140/250=

56%。因此这正确地解释了观察到的百分比就是最大似然值。

我们也可以计算平均数和中位数来概述后验概率：

```
print 'Mean', suite.Mean()
print 'Median', thinkbayes.Percentile(suite, 50)
```

均值为55.95，中位数为56。

最后，我们可以计算出一个置信区间：

```
print 'CI', thinkbayes.CredibleInterval(Suite, 90)
```

其结果是（51,61）。

现在，回到原来的问题，我们想知道硬币是否是均匀的。观察到的后验可信区间不包括50%，这就表明了硬币的确是不均匀的。

但确切地说，这不是开始的那个问题。Mckay提出的是“这些数据是否恰恰为——硬币偏心而非均匀——给出了证据？要回答这个问题，我们要更精确地理解“数据为某一假说提供了证据”这句话的含义。不过那是下一章的主题了。

那么，在继续探讨之前，我想先讲讲导致困扰的原因。既然我们要知道硬币是否均匀，很自然是求得*x* 为50%的概率：

```
print  suite.Prob (50)
```

结果是0.021，该值几乎说明不了什么。这使得对假设101的评估显得毫无意义。我们可将范围区间划分成更多或更少的细小区间，如果这样，对给定假设的概率则会更大或更小。

4.3 先验概率的湮没

本章开始，我们假设先验是均匀的，但这可能不是一个好选择。如

果硬币是偏心的，可以相信*x* 会大幅偏离50%，但如果偏心到使得*x* 是10%或90%就太不可能了。

更合理的是选择在 50%附近有较高概率，而在那些极端值上（指10%或90%）概率较低的一个先验。

作为一个例子，我构建了一个三角形状的先验概率，如图4-2所示。

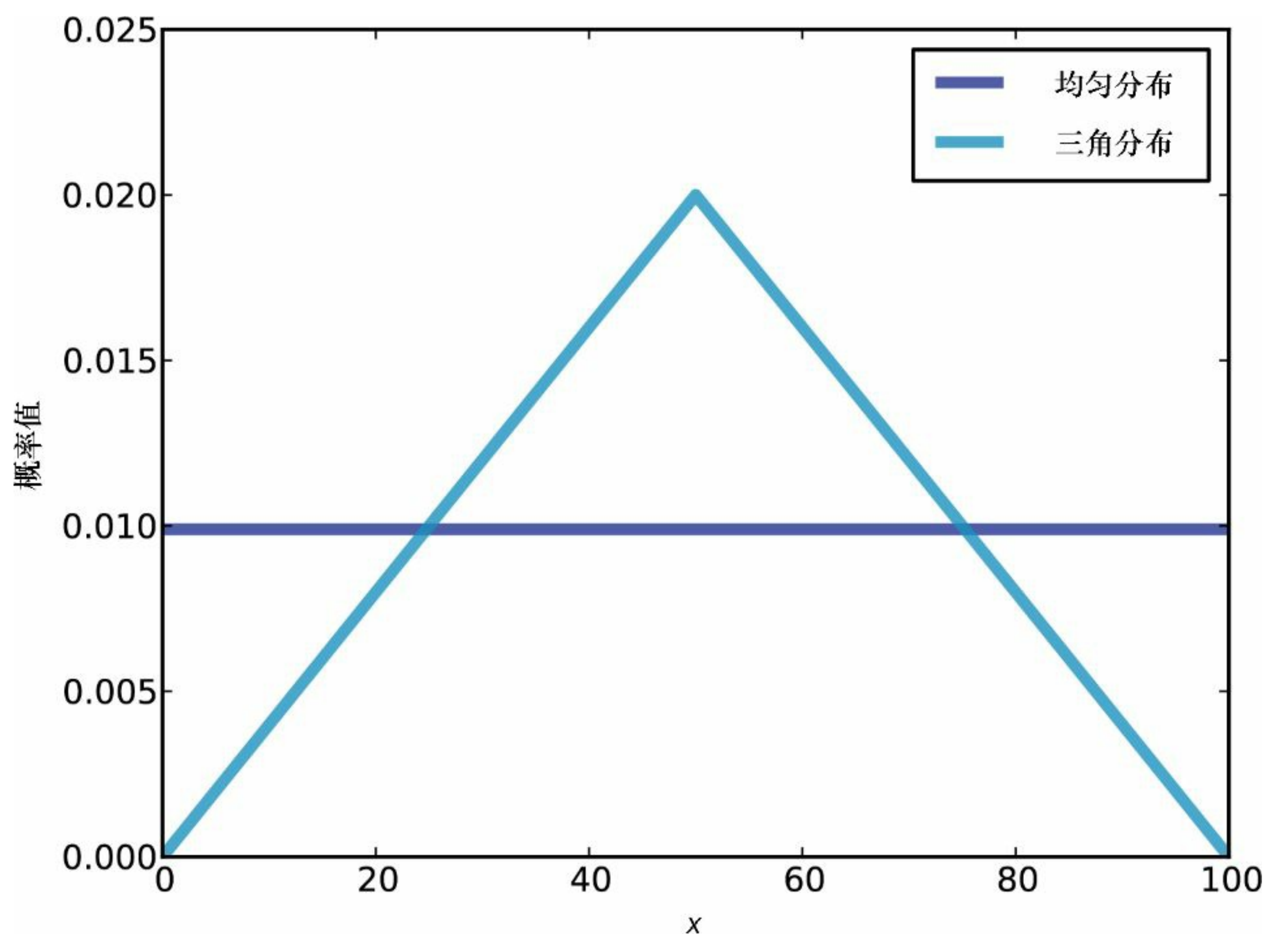


图4-2　欧元问题的均匀型和三角型分布先验

下面的代码构成了所述三角状的先验：

```
def TrianglePrior () :
    suite = Euro()
    for x in range(0 ,  51) :
        suite.Set (x, x)
    for x in range(51 ,  101) :
```

```
        suite.Set (x,  100 - x)
    suite.Normalize ()
```

图4-2显示了结果（和均匀先验概率比较）。以相同的数据集更新先验概率得到如图4-3所示的后验分布。即使实质上不同的先验，后验分布也非常相似。中位数和置信区间是相同的，均值差小于0.5%。

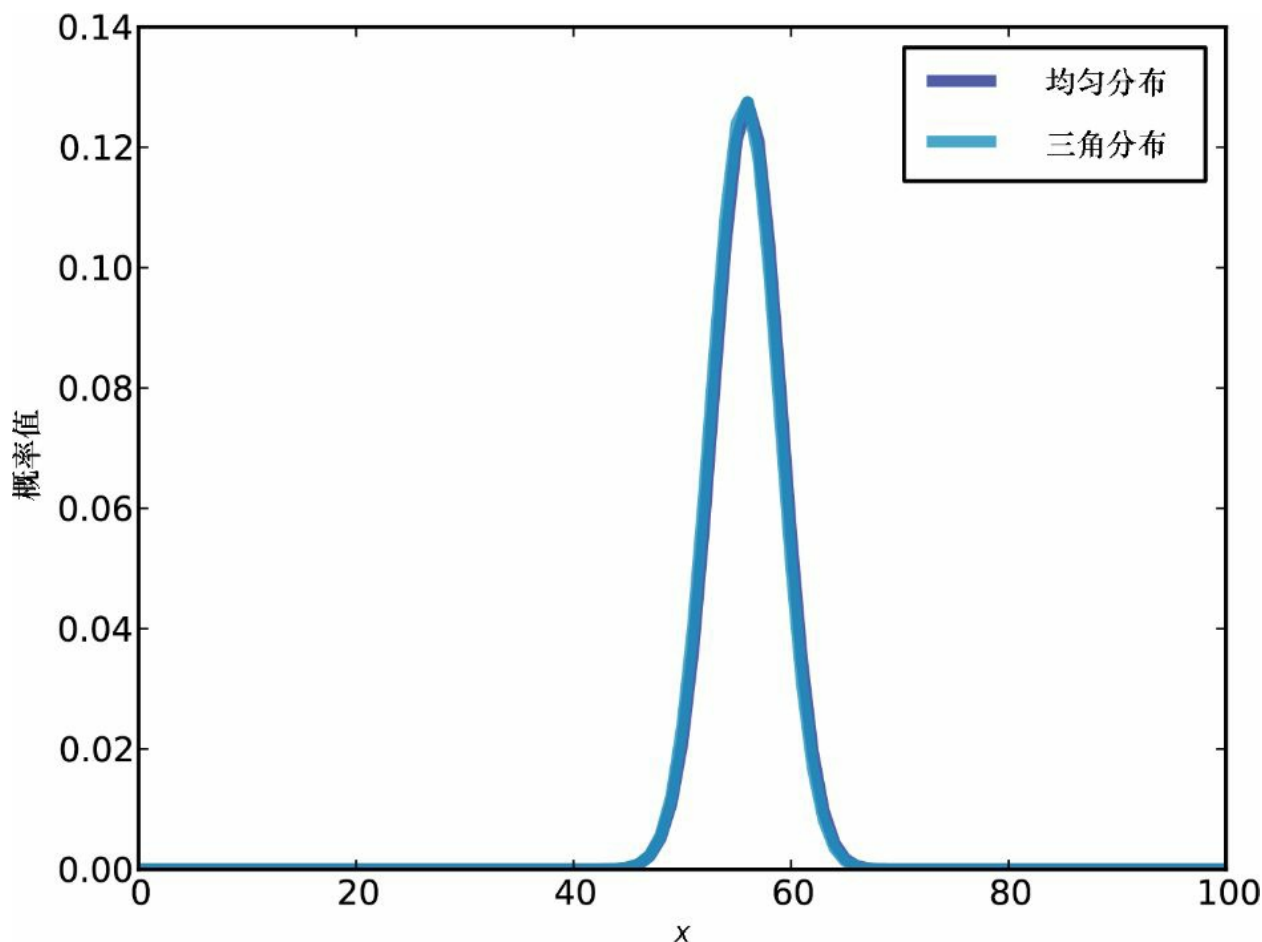


图4-3　欧元问题的后验分布

这就是先验湮没 的一个例子：如果有足够的数据，即使在先验分布上持有不同观点，人们也会得到趋于收敛的后验概率。

4.4　优化

到目前为止，我所展示的代码都是为了方便理解，但效率不算高。

通常，我先开发实证无误的代码，然后再检查对于达到目的是否够快。如果是，代码就没有进行优化的必要。

在这个例子中，如果我们关注运行时间，有几种方法可以加速代码运行。

第一个可能方法是减少归一化suite的次数。在原始代码中每一次模拟（转动硬币）都调用Update 一次。

```
dataset = 'H' * heads + 'T' * tails

for data in dataset:
    suite.Update(data)
```

下面是Update 方法：

```
def Update(self, data):
    for hypo in self.Values():
        like = self.Likelihood(data, hypo)
        self.Mult(hypo, like)
    return self.Normalize()
```

每次Update都会迭代假设集，然后调用Normalize，再次迭代假设集。我们可以通过在Normalize之前先update来节省一些时间。

suite 提供了一个名为UpdateSet 的方法就是这么实现的，如下：

```
    def UpdateSet(self, dataset):
        for data in dataset:
            for hypo in self.Values():
                like = self.Likelihood(data, hypo)
                self.Mult(hypo, like)
        return self.Normalize()
```

下面是我们怎么调用它：

```
    dataset = 'H' * heads + 'T' * tails
    suite.UpdateSet(dataset)
```

这优化了速度，但运行时间仍是与数据量成正比的。我们可以通过改写 Likelihood 处理整个数据集，而不是一次模拟运行一次。

在最初的版本中，数据编写为正面或反面的字符串：

```
def Likelihood(self, data, hypo):
    x = hypo / 100.0
    if data == 'H':
        return x
    else:
        return 1-x
```

有一种替代方法，我们可以将数据集当作有两个整数的元组来进行编码：正面和反面的数量。

在这种情况下的Likelihood 如下：

```
def Likelihood(self, data, hypo):
    x = hypo / 100.0
    heads, tails = data
    like = x**heads * (1-x)**tails
    return like
```

然后我们就可以像下面这样调用Update：

```
heads, tails = 140, 110
suite.Update((heads, tails))
```

由于我们用指数函数取代了重复的乘法，对任意次不同数量的模拟（转动硬币），这个版本代码消耗的运行时间是一样的。

4.5 Beta分布

还有一个进一步的优化让解法更快。

到目前为止，我们已经使用了Pmf对象来表示一组离散的x值。现在，我们将使用一个连续分布，确切地说是beta分布（见*http://en.wikipedia.org/wiki/Beta_distribution* ）。

beta分布定义在从0到1（包括两者）的区间上，所以它是一个描述比例和概率的自然选择。稍等，还要更好。

如果你像上节一样，用一个二项式似然函数来实现贝叶斯更新，加上beta分布是一个共轭先验 。那就意味着如果x的先验概率的分布是一个beta分布，那其后验概率也是一个beta分布。别急，比这还要好。

beta分布的形状取决于两个参数，写为α 和β ，或`alpha` 和`beta` 。如果先验概率是带有参数`alpha` 和`beta` 的beta分布，我们看到`h` 次正面和`t` 次反面的数据，后验概率就是参数为`alpha+h` 和`beta+t` 的beta分布。换句话说，通过两个加法实现了Update方法。

所以这很了不起，只不过仅适用于先验概率的分布的确是一个beta分布的情形。幸运的是，在最低限度上，对许多实际的先验分布beta分布都可以进行良好的近似，同时也可以完美匹配均匀前验。参数`alpha=1` 和`beta=1` 的beta分布就是从0到1的均匀分布。

让我们看看如何利用这一切优势。`thinkbayes.py` 提供了一个类来表示beta分布：

```
class Beta(object):

    def __init__(self, alpha=1, beta=1):
        self.alpha = alpha
        self.beta = beta
```

默认情况下的`__init__` 使用均匀分布。`Update` 进行贝叶斯更新：

```
    def Update(self, data):
        heads, tails = data
        self.alpha += heads
        self.beta += tails
```

`data` 是一对代表正面和反面数量的整数。

因此，我们有另一种方法来解决欧元问题：

```
beta = thinkbayes.Beta()
beta.Update((140, 110))
print beta.Mean()
```

beta 提供了Mean ，计算alpha 和beta 的一个简单函数：

```
def Mean(self):
    return float(self.alpha) / (self.alpha + self.beta)
```

对于欧元问题后验的平均值为56%，这和我们使用Pmf得到的结果是相同。

beta 还提供了EvalPdf ，它计算beta分布的概率密度函数（PDF）：

```
def EvalPdf(self, x):
    return x**(self.alpha-1) * (1-x)**(self.beta-1)
```

最后，beta 提供MakePmf 方法，它使用EvalPdf 生成beta分布的离散近似值。

4.6 讨论

在本章中，我们用两个不同的先验处理同一个问题，发现在较大数据集的条件下，先验之间的区别被掩盖了。如果两个人开始之前对于先验有不同信仰，他们通常会发现，随着得到更多的数据，后验分布收敛了。在一些点上，分布之间的差异足够小到没有实际影响。

当这种情况出现时，会减轻一些我在前一章关于客观性的担忧。甚至对于许多现实世界的问题，明显不同的先验信念最终都会被数据所矫正。

但事实并非总是如此。首先，请记住，所有的贝叶斯分析是基于模

型决策的。如果你和我没有选择相同的模型，我们可能对数据进行不同的解读（诠释）。因此，即使使用相同的数据，我们也会计算得到不同的似然度，因而后验概率就可能不收敛。

另外，请注意，在贝叶斯Update中，我们以一个似然度乘以每个先验概率，所以如果p(H)为0，p(H|D)也为0，而不论D是什么。在欧元问题上，如果你确信x小于50%，而且指定其他所有假设的概率为0，那么再多的的数据都将无法说服你。

这种看法是克伦威尔法则 的基础，建议是：应当避免设置任何一个假设的先验概率为0，哪怕的确存在这种可能性（见 *http://en.wikipedia.org/wiki/Cromwell's_rule* ）。

克伦威尔法则以奥利弗·克伦威尔命名，他写下了"我求求你，看在基督的份上，认为这可能是你是误会了"。对于贝叶斯方法，这也是个好建议（即使有点过度重视了）。

4.7 练习

练习4-1。

假设不是直接观察掷硬币，而是使用一个不总是正确的工具来测量结果。具体地说，假设有一个概率y 将实际正面报告为反面，或报告实际反面为正面。

编写一个类来估计这样一系列给定了结果的硬币的偏置量和值y，后验分布的范围是怎样依赖于y 的？

练习4-2。

这项练习受到Reddit上一个帖子的启发，一位名叫dominosci 的"redditor"在Reddit的统计小组上提出了这个问题。*http://reddit.com/r/statistics* 。

Reddit是一个在线论坛，有许多被称为的subreddits的兴趣小组。用户被称为redditors，redditor会贴上网上内容和其他网页的链接。其他redditors对此链接进行表决，对高质量的链接给予"upvote"，不好或者是不相关的链接则给予"downvote"。

Dominosci提出了这样一个问题，就是redditor当中一些人比另一些人更可信，但是Reddit并没有考虑到这个因素。

我们的挑战是设计一个系统，当redditor做了投票，可以根据redditor的可信度估计并且更新链接的质量，而对该redditor的可信度估计也会依据这一链接的质量被更新。

一种方法是将链接的质量以赢得upvote的概率建模，redditor的可信度以对高品质条目给出正确upvote的概率建模。

编写一个类来定义redditors和链接，只要redditor投了票，update函数对这两个对象进行更新。

第5章　胜率和加数

5.1　胜率

表示概率的方法之一是用0和1之间的数字，不过这并非唯一的方法。如果你玩过足球博彩或赛马，可能遇到概率的另一种表示，称为胜率（**odds**）[①]。

你应该听说过例如“胜率是三比一”这样的表述，但也许不知道含义。胜率是一个事件可能发生的概率与不发生的概率的比值。

所以，如果认为我的球队有75％的机会获胜，我会说他们的胜率是三比一，因为获胜的机会三倍于失败的机会。

你可以将胜率写成十进制形式，但最常见的是将其写成整数比。因此“三比一”被写成3:1。

当概率较低，通常称为赔率（**odds against**），而不是胜率（**odds in favor**）。例如，如果觉得我的马只有10%获胜的机会，我会说赔率是9:1。

概率和胜率是相同信息的不同表示形式。给定一个概率，你可以这样计算胜率：

```
def Odds(p):
    return p / (1-p)
```

给定（支持）胜率的十进制形式，可以这样将其转换为概率：

```
def Probability(o):
    return o / (o+1)
```

如果你以分子和分母表示胜率，可以像下面这样进行概率转换：

```
def Probability2(yes, no):
    return yes / (yes + no)
```

当带着胜率思维工作时，我发现它有助于确定人们的观点。如果20％的人认为我的马赢，那么其余80％的人当然就相反，因而赞成赔率是20:80或1:4 。

如果我的马赢的反向赔率是5:1，那么就是六分之五的人认为它会失败，所以获胜的概率是1/6。

5.2 贝叶斯定理的胜率形式

在第1章我写出了贝叶斯定理的概率形式：

$$\mathrm{p}(H|D)=\frac{\mathrm{p}(H)\mathrm{p}(D|H)}{\mathrm{p}(D)}$$

如果我们有A 和B 两个假设，我们可以写出后验概率的比值如下：

$$\frac{\mathrm{p}(A|D)}{\mathrm{p}(B|D)}=\frac{\mathrm{p}(A)\mathrm{p}(D|A)}{\mathrm{p}(B)\mathrm{p}(D|B)}$$

请注意等式中出现的标准化常数p (D) 。

如果A和B是互斥且穷尽的，就意味着p(B) = 1−p (A)，因此我们可以将先验的比率、后验的比率重写为胜率。将支持A 的可能性写为o(A)，得到：

$$\mathrm{o}(A|D)=\mathrm{o}(A)\frac{\mathrm{p}(D|A)}{\mathrm{p}(D|B)}$$

在字面形式上，这说明了后验赔率是先验胜率乘以似然比。而这正是贝叶斯定理的胜率形式。

这种形式相当适合在纸上或者脑海中进行贝叶斯计算。

例如，我们回到曲奇饼问题：

假设有两碗曲奇饼。碗1包含30个香草曲奇饼和10个巧克力曲奇饼。碗2包含两种曲奇饼各20个。

现在假设你随意选择了一个碗，然后随意选择一个曲奇饼。如果是香草曲奇饼，它来自碗1的概率是多少？

先验概率是50％，所以胜率是1:1，或就是1。似然度是$\frac{3}{4}/\frac{1}{2}$，或3/2。所以后验概率就是3:2，对应于概率3/5。

5.3　奥利弗的血迹

下面是来自麦凯的《信息理论、推理和学习》的另一个问题。

在一个犯罪现场，有两人遗留了血迹。一名疑犯奥利弗经过测试发现是“O”型血。而发现的痕迹中血型分别是“O”型（一种本地人口的常见血型，有60％的概率）和“AB”型（一种罕见的血型，概率1％）。

那么这些数据[现场发现的痕迹]是否支持奥利弗是疑犯之一[在现场遗留下血液证据的人]?

要回答这个问题，我们需要想想“数据支持了假设”这到底意味着什么。直觉上，相比之前如果某一假设随着数据的出现而呈现更大可能性，我们就说“数据支持了假设”。

在曲奇饼问题上，先验胜率是1:1，或者概率50％。后验胜率是3:2，或者概率60％。因此，我们可以说，香草曲奇饼这一数据作为证据支持其来自碗1的假设 。

贝叶斯定理的胜率形式提供了一种方法，使这种直觉更准确。

回顾一下公式

$$\mathrm{o}(A|D) = \mathrm{o}(A)\frac{\mathrm{p}(D|A)}{\mathrm{p}(D|B)}$$

或除以o(*A*)：

$$\frac{\mathrm{o}(A|D)}{\mathrm{o}(A)} = \frac{\mathrm{p}(D|A)}{\mathrm{p}(D|B)}$$

等式左边是后验胜率和前验胜率的比值。右边是似然比，也称为贝叶斯因子 。

如果贝叶斯因子的值大于1，则意味着数据更可能支持假设*A* 而不是假设*B*，也意味着胜算更大，鉴于胜率比也大于1，意味着随着数据的出现胜率也增加了。

如果贝叶斯因子小于1，就意味着数据支持*B* 的可能大于支持可能*A*，所以支持*A* 的胜率降低了。

最后，如果贝叶斯因子恰好为1时，数据或者假说有同等可能，所以胜率不会改变。

现在我们可以回到奥利弗血迹问题了。如果奥利弗是在犯罪现场留下血迹的人之一，就解释了那个“O”型血证据样本的由来，因此数据的概率就是在人群中随意挑中一个“AB”血型人的概率1％。 ②

如果奥利弗没有在现场留下血液，我们就要对这两个样本进行解释。假设在人群中任选两人，有多大可能性正好找到一个“O”血型人和一个“AB”血型人？嗯，两种可能情况：第一个人是“O”型第二个是“AB”型，或者反过来组合一下。所以总的概率是2 (0.6)(0.01) = 1.2％。

如果不是奥利弗的血液，数据的似然度还要稍高些，所以血液证据这个数据实际上没有支持奥利弗的犯罪嫌疑。

这个例子有刻意设计的成分，但它是一个违反我们自觉的、符合假设的数据却并非必然支持假设的例子。

如果这一结果如此有悖常理甚至困扰到你，下面的思路可能有些帮助：该数据由一个常见事件——“O”型血，和一个罕见事件 ——“AB”型血构成。如果奥利弗与常规事件相关（O型血），这使得罕见的事件还是无法解释。如果奥利弗与常规事件无关，那么我们有两种可能找到“AB”型血的疑犯。两种情况中的这一因素导致了差别。

5.4 加数

贝叶斯统计的基本操作是Update，这需要先验概率和一组数据，并产生一个后验分布。但是，解决实际问题通常涉及许多其他操作，包括缩放、加法和其他算术运算、最大值和最小值，还有混合计算。

本章介绍加法和最大值；在需要的时候我会介绍其他的运算。

第一个例子基于《龙与地下城》这个角色扮演游戏，在这个游戏里，玩家决策的结果是通过掷骰子来完成的。

事实上，游戏开始前，通过转动3个6面骰子并把结果相加得到一个总和，玩家创造了自己角色的各个属性）力量、智力、英明、灵巧、体质和魅力。

所以，你应该会对总和的分布感兴趣。有两种方法可以计算。

模拟：

给定一个表示骰子面分布的Pmf，可以绘制随机样品，把它们加起来，累加每次模拟求和的分布。

枚举：

给定两个Pmfs，可以枚举所有可能的数值对，并计算和的分布。

`thinkbayes` 提供这两个函数。下面是第一个方法的实现。首先，我定义一个类来表示单个骰子：

```
class Die(thinkbayes.Pmf):

    def __init__(self, sides):
        thinkbayes.Pmf.__init__(self)
        for x in xrange(1, sides+1):
            self.Set(x, 1)
        self.Normalize()
```

现在，可以创建一个6面骰子：

```
d6 = Die(6)
```

然后使用thinkbayes.SampleSum 产生1000次转动3个骰子的样本。

```
dice = [d6] * 3
three = thinkbayes.SampleSum(dice, 1000)
```

SampleSum 以分布的列表（Pmf或Cdf对象）和样本大小n 为参数。它产生n 次随机样本的和（单次模拟的汇总——转动3次骰子），并将其分布作为一个Pmf对象返回。

```
def SampleSum(dists, n):
     pmf = MakePmfFromList(RandomSum(dists) for i in xrange(n))
     return pmf
```

SampleSum 使用RandomSum，也在thinkbayes.py 中：

```
def RandomSum(dists):
    total = sum(dist.Random() for dist in dists)
    return total
```

RandomSum 在每个分布中调用Random 然后汇总其结果。

模拟的缺点是得到的结果只是近似正确的，随着n 变大，结果会更准确，但是运行时间也增加了。

另一种方法是枚举所有成对的值，并计算每对的概率和总和。这在Pmf.__add__ 上实现：

```
# class Pmf

     def __add__(self, other):
         pmf = Pmf()
         for v1, p1 in self.Items():
             for v2, p2 in other.Items():
                 pmf.Incr(v1+v2, p1*p2)
```

```
        return pmf
```

`self` 是一个Pmf对象，`other` 可以是Pmf或其他任何提供条目的对象。其结果是一个新的Pmf对象。运行`__add__` 的时间取决于`self` 和`other` 对象中的条目的数量，它正比于`len(self)*len(other)` 。

下面是如何使用它：

```
three_exact = d6 + d6 + d6
```

当在Pmf中应用+操作符时，Python调用`__add__` 。在这个例子中，`__add__` 被调用了两次。

图5-1显示了通过模拟生成的近似结果和枚举计算产生的确切结果。

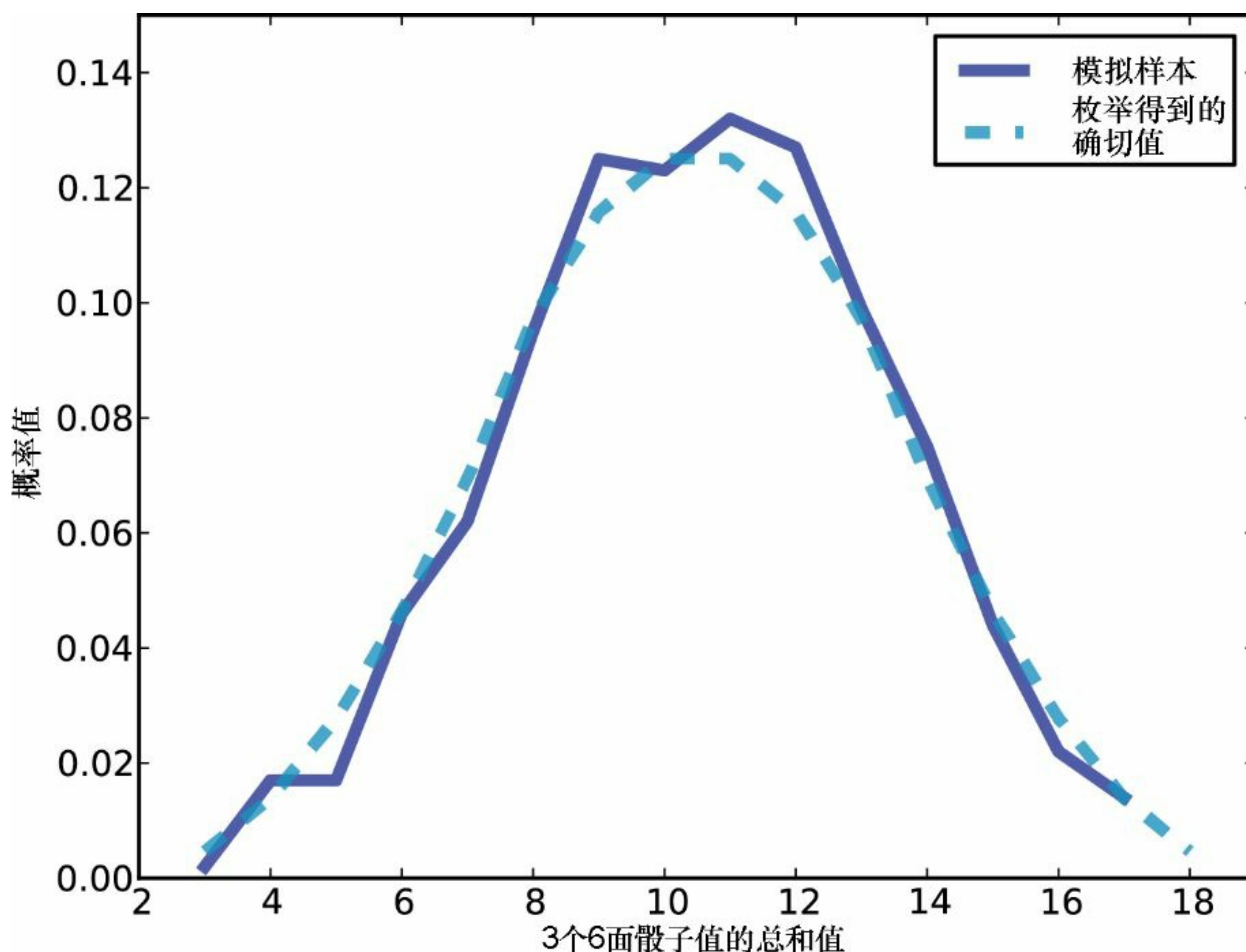


图5-1　转动3个6面骰子总和值的近似和确切分布

`Pmf.__add__` 是基于这样的假设，来自每个Pmf的随机选择是独立的。在投掷多个骰子的例子中，这个假设相当不错。其他情况下我们则必须扩展这个方法以使用条件概率。

从本节中的代码可以从 *http://thinkbayes.com/dungeons.py* 获得。更多信息请参见前言的“代码指南”。

5.5　最大化

当你生成一个《龙与地下城》的角色，会对人物的最佳属性特别有兴趣，所以你可能想知道属性值分布的最大值。

有三种方法来计算一个最大值的分布。

模拟：

给定一个Pmf，代表单一选择的分布，可以生成随机样本，找到最大值和模拟最大值的累积分布。

枚举：

给定两个Pmf，可以枚举所有可能的数值对，并计算分布的最大值。

指数计算：

如果我们将一个Pmf转换为Cdf，有一个简单而有效的算法查找最大的Cdf（看后面）。

模拟最大值的代码与模拟求和的代码几乎相同：

```
def RandomMax(dists):
    total = max(dist.Random() for dist in dists)
    return total
def SampleMax(dists, n):
    pmf = MakePmfFromList(RandomMax(dists) for i in xrange(n))
    return pmf
```

我所做的只是用“max”替换“sum”。而对于枚举部分代码几乎是相同的：

```
def PmfMax(pmf1, pmf2):
    res = thinkbayes.Pmf()
    for v1, p1 in pmf1.Items():
        for v2, p2 in pmf2.Items():
            res.Incr(max(v1, v2), p1*p2)
    return res
```

事实上，你可以将操作符作为参数来一般化这个函数。

这一算法的唯一问题是，如果每个Pmf具有m 个值，运行时间正比于m^2 。如果我们想知道k 个选择的最大值，需要的时间正比于km^2 。

如果我们转换Pmfs到Cdfs，我们可以以更快的速度进行同样的计算！关键是要记住累积分布函数的定义：

$$CDF(x) = \mathrm{p}(X \leqslant x)$$

其中*X* 是一个随机变量，它的意思是“从分布中随机选取的一个值”，所以举例来说，*CDF* (5)表示该分布中某个值是小于或等于5的概率。

如果我从CDF_1中取出*X*，从CDF_2中取出*Y*，计算最大*Z* =*max* (*X*，*Y*)，则*Z* 小于或等于5的可能性是多少？当然，在这个案例里面，*X* 和*Y* 必须小于或等于5。

如果选择*X* 和*Y* 是独立行为，

$$CDF_3(5) = CDF_1(5)CDF_2(5)$$

其中CDF_3是*Z* 的分布。我选择了5这个值让公式容易阅读，但我们可以一般化为*z* 是任意值的情况：

$$CDF_3(z) = CDF_1(z)CDF_2(z)$$

有一个特例是我们从同一个分布中选择*k* 值，

$$CDF_k(z) = CDF_1(z)^k$$

因此，为了求*k* 的最大值的分布，我们可以枚举给定Cdf的概率再将其变化为*k* 次幂。`Cdf` 提供了一种方法，该方法如下：

```
# class Cdf

    def Max(self, k):
        cdf = self.Copy()
        cdf.ps = [p**k for p in cdf.ps]
        return cdf
```

`Max` 方法接受选取的次数`k`，然后返回一个新的表示进行`k` 次选择最大值的Cdf。此方法的运行时间正比于Cdf中的条目个数*m* 。

`Pmf.Max` 和Pmfs实现同样的功能。只是将Pmf转换到Cdf必须要多做一点工作，所以运行时间成正比于*mlogm*，但是这仍然要好过于呈平方复杂度的运行时间。

最后，这里有一个计算角色的最佳属性的分布的范例：

```
best_attr_cdf = three_exact.Max (6)
best_attr_pmf = best_attr_cdf.MakePmf ()
```

其中`three_exact` 是上一节定义过的。如果我们打印出结果，将看到产生一个有18属性值的角色的概率大约是3%。

图5-2显示了分布。

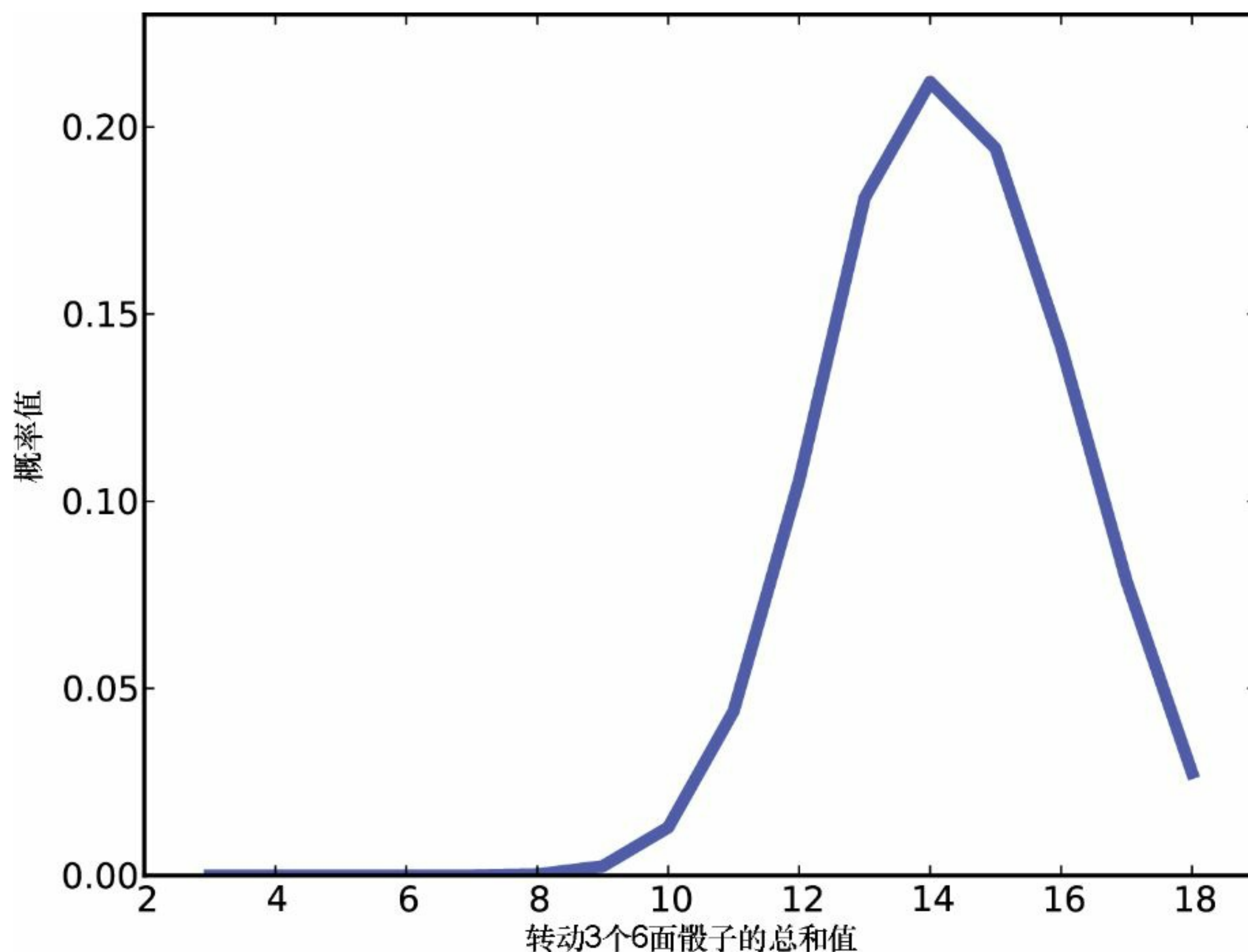


图5-2　转动3个6面骰子的最大值分布

5.6　混合分布

让我们考虑一个来自《龙与地下城》的例子。假设我有一盒骰子，清单如下：

```
5个      4面骰子
4个      6面骰子
3个      8面骰子
2个     12面骰子
1个     20面骰子
```

从盒子中选择一个骰子转动，结果会呈现什么分布呢？

如果你知道选到的是哪个骰子，答案很简单。n 面的骰子产生一个均匀的从1到n 的分布，包括1和n 。

但是，如果我们不知道选到的究竟是哪个骰子，由此产生的分布则是有不同上界的均匀分布的混合分布 。

在一般情况下，这类混合量不适合任何简单的数学模型，但它可以从对PMF的直接计算得到。

与往常一样，一个选择是模拟，生成一个随机样本计算模拟样本的PMF。这种方法很简单，它会快速得到一个近似的结果。但是，如果想要确切的解法，我们需要一种不同的方法。

让我们从只有两个骰子的简单版本开始，一个6面骰和一个8面骰。我们可以用Pmf来表示每个骰子：

```
d6 =Die(6)
d8 =Die (8)
```

然后我们创建一个Pmf表示这一混合分布：

```
mix = thinkbayes.Pmf()
for die in [d6, d8]:
    for outcome, prob in die.Items():
        mix.Incr(outcome, prob)
```

```
mix.Normalize()
```

第一个循环枚举骰子，第二个循环枚举循环骰子结果和概率。在循环内部，`Pmf.Incr` 汇总了两个分布的贡献。

此代码假定选中两个骰子是同等可能的。更一般地，我们需要知道选中每一个骰子的概率，以便可以对结果进行对应的加权。

首先，我们创建一个映射了骰子和骰子被选中概率的Pmf对象：

```
    pmf_dice = thinkbayes.Pmf()
    pmf_dice.Set(Die(4), 2)
    pmf_dice.Set(Die(6), 3)
    pmf_dice.Set(Die(8), 2)
    pmf_dice.Set(Die(12), 1)
    pmf_dice.Set(Die(20), 1)
    pmf_dice.Normalize()
```

接下来，我们需要计算混合分布算法的一个更通用的版本：

```
    mix = thinkbayes.Pmf()
    for die, weight in pmf_dice.Items():
        for outcome, prob in die.Items():
            mix.Incr(outcome, weight*prob)
```

现在每个骰子都有相关联的权重了（意味着骰子有了权值）。

当将每个结果添加到混合分布中，其概率乘以权重，得到了图5-3所示的结果。正如预期的那样，1至4是最有可能的，因为每一个骰子都可以生成这些值。12以上的值可能性不大，因为只有一个20面骰子能有大于12的值（即使是20面骰子，要产生这样值的可能性也都不到一半）。

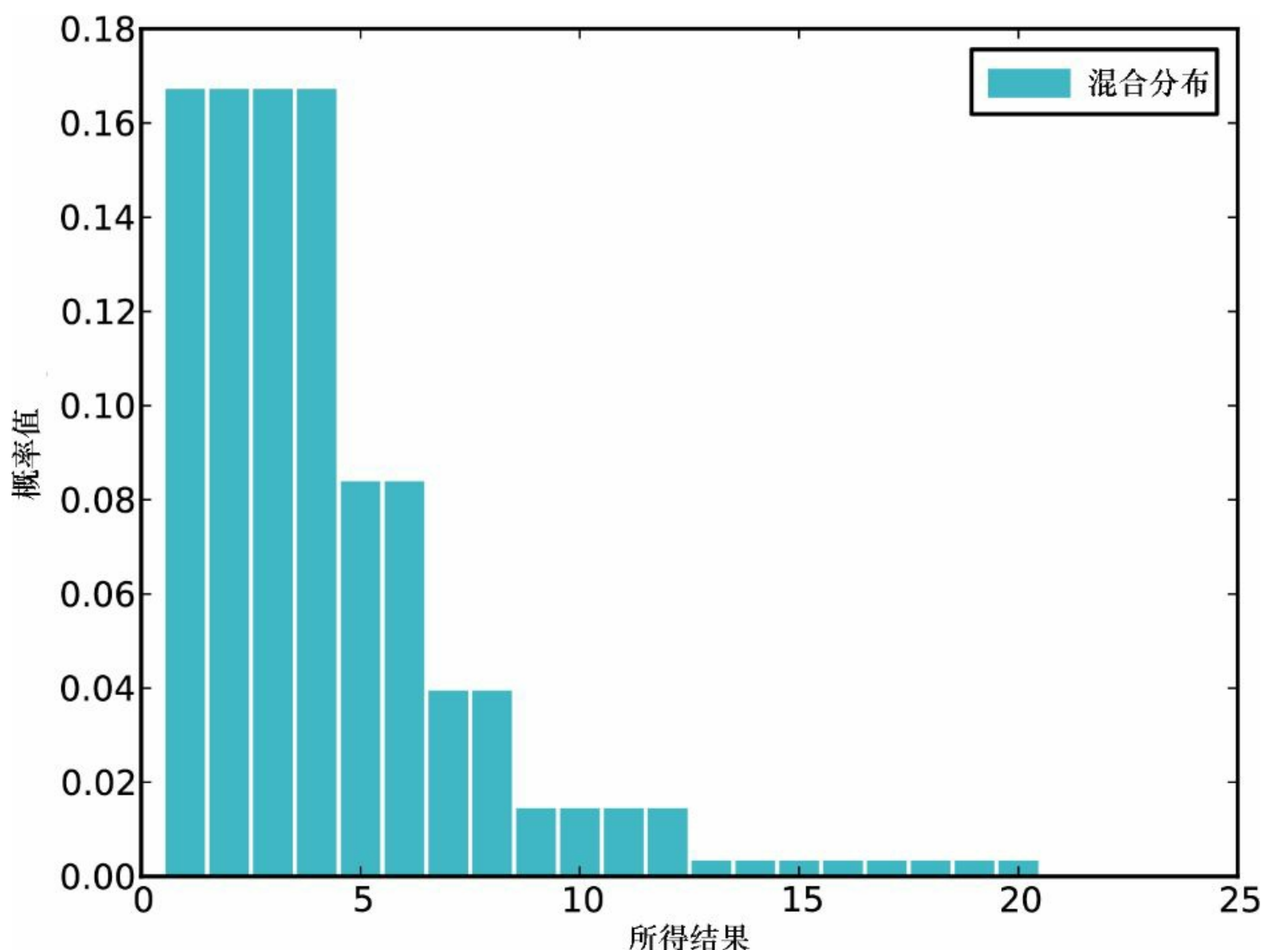


图5-3　从盒子中随机选中骰子的结果分布

thinkbayes 提供了一个名为MakeMixture 的函数，它封装了该算法的功能，所以我们把计算过程写成：

```
mix = thinkbayes.MakeMixture (pmf_dice)
```

我们将在第7章和第8章再次使用MakeMixture 。

5.7　讨论

除了贝叶斯定理的赔率形式，这一章不是专门关于贝叶斯的。但贝叶斯分析的一切都和分布有关，所以了解分布的概念很重要。以计算的观点来看，分布就是可以表示任意一组值（一个随机过程的可能结果）和其概率的数据结构。

我们已经看到了分布的两种表示形式：Pmfs和Cdfs。在包含了相同信息这一点上，它们是等价的，这样你就可以从一个转换到另一个。它们之间的主要区别在于性能：有些运算采用Pmf更快、更容易，另一些则是采用Cdf更快。

本章的另一个目标是引入一些概率分布运算操作，像`Pmf.__add__`，`Cdf.Max`和`thinkbayes.MakeMixture`。后续我们将使用这些运算，但这里，我介绍的目的是鼓励大家将分布作为概率计算的基石进行深入思考，而不仅仅是将其当作一个值和概率的容器。

---

① 译注：odds经常被直观但不严谨地称为赔率（这与博彩业有关，博彩业就某一事件的赔率严格说是大于1的，否则不赚钱），事实上odds是机会的意思。

② 译注：记住数据包括两个因素，O型血和AB型血的现场证据。

第6章　决策分析

6.1　“正确的价格”问题

2007年11月1日，选手莱希娅和纳撒尼尔参加《正确的价格》这个美国电视游戏节目。他们在名为showcase的游戏环节中比赛，游戏主题是猜展示奖品的价格。猜测到最接近展示品实际价格的选手将赢得奖品。

纳撒尼尔先开始。他的展品里面包括洗碗机、酒柜、笔记本计算机和一辆汽车。他出价26000美元。

莱希娅的展品包括弹球机、视频游戏、台球桌和一次去巴哈马的旅行。她出价21500美元。

纳撒尼尔展品的实际价格为25347美元。由于出价太高他输掉了比赛。

莱希娅的展品的实际价格为21578美元。比起实际价格她只少猜了78美元所以赢得了比赛。而且因为她的出价差少于250美元，她还赢得了纳撒尼尔的展品。

对一个具备贝叶斯思维的人，这一场景暗示出的几个问题是：

1．在看到奖品前，选手对展示品的价格应该有什么样的先验分布判断？

2．看到奖品后，选手应该如何修正这些预期？

3．基于后验分布，选手应该怎么出价？

第三个问题，论证了贝叶斯分析的一个常见用途：决策分析。给定一个后验分布，我们可以选择出价多少，从而最大限度地提高选手的预期收益。

这个问题来自卡梅伦·戴维森-皮隆的《Bayesian Methods

forhackers》一书。我为本章书写的代码可从*http://thinkbayes.com/price.py* 得到；输入的数据可以从*http://thinkbayes.com/showcases.2011.csv* 和*http://thinkbayes.com/showcases.2012.csv* 下载。更多信息请参考前言的“代码指南”。

6.2　先验概率

为了选择价格的先验分布，我们可以利用先前的数据。幸运的是，这个节目的粉丝详细记录了这些数据。当我与戴维森-皮隆先生就本书通信时，他提供了史蒂夫·吉收集的数据 *http://tpirsummaries.8m.com* 。它包括了从2011年到2012年的节目中每个展品环节的价格，还有参赛选手就展品提供的出价。

图6-1显示了这些展品价格的分布。最常见的展品价值大约是28000美元，但是第一组展品在50000美元附近有个第二特征（线图稍微隆起的地方），第二组展品的价格偶尔会超过70000美元。

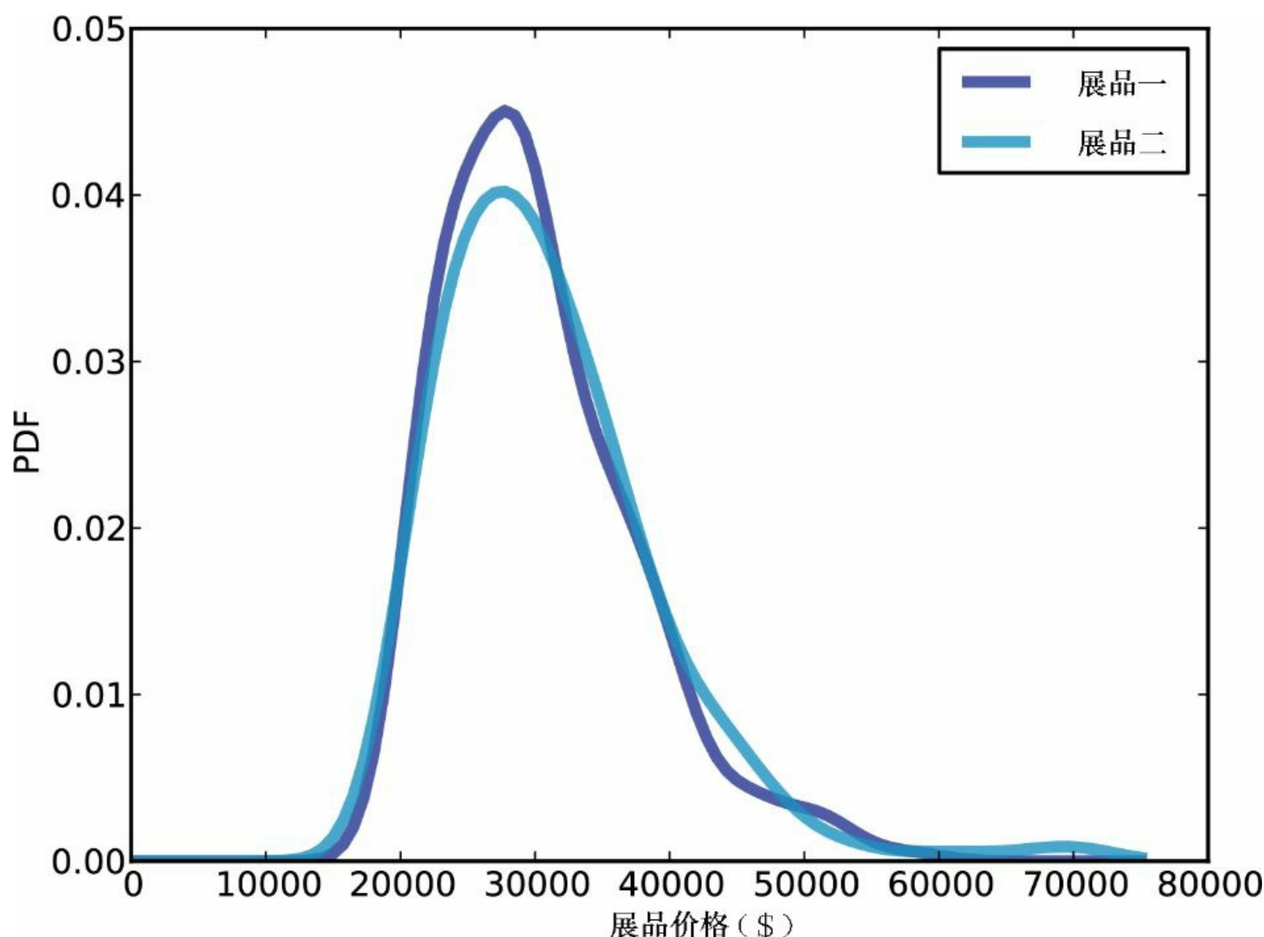


图6-1 “正确的价格”节目中展品价格的分布，2011-2012

这些分布基于实际的数据，但用高斯内核密度估计进行了平滑（KDE）。在我们继续之前，我想绕道先去谈谈概率密度函数和KDE。

6.3 概率密度函数

到目前为止，我们已经和概率质量函数或PMF打了很多交道。PMF是从每一个可能值到其概率的映射。在我的实现中，Pmf对象提供一个`Prob` 方法获得值和概率，也被称为概率质量。

在数学表示上，PDF通常写成一个函数；例如，这里是均值为0，标准偏差为1的高斯分布的PDF：

$$f(x)=\frac{1}{\sqrt{2\pi}}\exp(-x^2/2)$$

对于一个给定的*x* 值，这个函数可以计算出概率密度。在更高的密度说明可能性也高这个意义上，密度和概率质量是相似的。

但是密度不是概率。密度可以是0或任何正值；并非如概率那样在0和1之间有界。

如果你在一个连续区间对密度进行积分，其结果是一个概率。但在本书中，我们很少这么用。

相反，我们主要利用概率密度作为似然函数的一部分。下面马上会看到一个例子。

6.4 PDF的表示

要在Python中表示PDF，`thinkbayes.py` 提供一个名为`PDF` 的类。`PDF` 是一个抽象类，这意味着它定义了PDF接口，但不提供一个完整的实现。PDF接口包括两种方法，`Density` 和`MakePmf`：

```
class Pdf(object):

   def Density(self, x):
        raise UnimplementedMethodException()

   def MakePmf(self, xs):
        pmf = Pmf()
        for x in xs:
             pmf.Set(x, self.Density(x))
        pmf.Normalize()
        return pmf
```

`Density` 取一个`x` 值，并返回相应的密度。`Makepmf` 生成PDF的近似离散值。

`Pdf` 提供了`makepmf` 的实现，而没有`Density` 方法，`Density` 方法必须通过子类来提供。

实体类 是继承自抽象类的一个子类，它提供了缺失方法的具体实现。例如，`GaussianPdf` 扩展了`Pdf` 并提供了`Density` 方法：

```
class GaussianPdf(Pdf):

    def __init__(self, mu, sigma):
        self.mu = mu
        self.sigma = sigma

    def Density(self, x):
        return scipy.stats.norm.pdf(x, self.mu, self.sigma)
```

`__init__` 接受`mu` 和`sigma` 参数，分别代表分布的平均值和标准偏差，并作为属性值存下来。

`Density` 使用`scipy.stats` 中的一个函数来评估高斯PDF。这个函数是`norm.pdf`，因为高斯分布也称为“正态”分布。

高斯PDF由一个简单的数学函数定义，所以容易求值。而且由于现实世界中大量的分布都可以近似为高斯分布，它的用处也很大。

但对于真实数据，并不能保证分布是高斯或任何其他的简单数学函数。在这种情况下，我们可以用一个样本来估计整体的PDF。

例如，“正确的价格”中的数据，我们有313个第一组展品的价格。我们可以认为这些值是展品价格的一个样本。

该示例包括以下值（按顺序）：

28800, 28868, 28941, 28957, 28958

在样品中，没有出现28801和28867之间的值，但是没有理由认为这些值不存在。基于我们的背景信息，我们希望在这个范围内的所有值有同等可能。换句话说，我们预计PDF是非常平滑的。

内核密度估计（KDE）是一种算法，从样本找到一个恰当平滑的PDF进行数据拟合。你可以从 http://en.wikipedia.org/Wiki/Kernel_density_estimation 了解到一些细节。

`Scipy` 提供了KDE的实现，`thinkbayes` 则提供了一个`EstimatedPdf` 类使用它：

```
class EstimatedPdf(Pdf):

    def __init__(self, sample):
        self.kde = scipy.stats.gaussian_kde(sample)

    def Density(self, x):
        return self.kde.evaluate(x)
```

__init__ 采集样本计算内核密度估计。其结果是一个gaussian_kde 对象，并提供了一个evaluate 方法。

Density 接收数据，调用gaussian_kde.evaluate ，然后返回密度结果。

最后，这是生成的代码主干：

```
    prices = ReadData()
    pdf = thinkbayes.EstimatedPdf(prices)

    low, high = 0, 75000
    n = 101
    xs = numpy.linspace(low, high, n)
    pmf = pdf.MakePmf(xs)
```

pdf 是一个由KDE估算的Pdf对象，pmf 则是一个近似于Pdf的Pmf对象，它通过在成序列的等间距区间上的求密度值实现。

linspace 表示“线性空间”的意思。它接收一个由low 和high 界定的区间，以及区间内的n 个数据点，并返回一个新的numpy 数组，该数组包含了n 个low 和high 间的等间距元素。

现在回到“正确的价格”问题。

6.5　选手建模

图6-1的PDF估计了展品可能的价格分布。如果你是一个参加节目的选手，你可以使用这个分布来量化每次展品环节中的先验信念（在看到奖品前）。

要更新这些先验概率，我们必须回答下面这些问题：

1．怎么看待数据以及如何量化数据？

2．怎么计算似然函数？即，对于每个价格给出什么样的假设值，怎么计算数据的条件似然度？

为了回答这些问题，我将选手作为一种误差特性已知的价格猜测仪来建模。换句话说，当选手看到每组展品猜单个奖品的价格时——不考虑奖品是展品一部分这一事实（也就是不考虑总量）——再把这些价格加起来，得到的总和称为猜测价格guess 。

在这一模型下，我们必须要回答的问题是 “如果实际价格是price ，选手的估计价格就是猜测价格guess 的似然度？”

如果我们定义：

```
error = price – guess（猜测误差 = 展品价格 – 猜测价格）
```

然后，我们可以提出问题“选手的估计价格背离猜测误差error 的似然度是什么？”

要回答这个问题，我们要再次使用历史数据。图6-2显示了diff 的累积分布，diff 是参赛者的出价和展示的实际价格间的差。

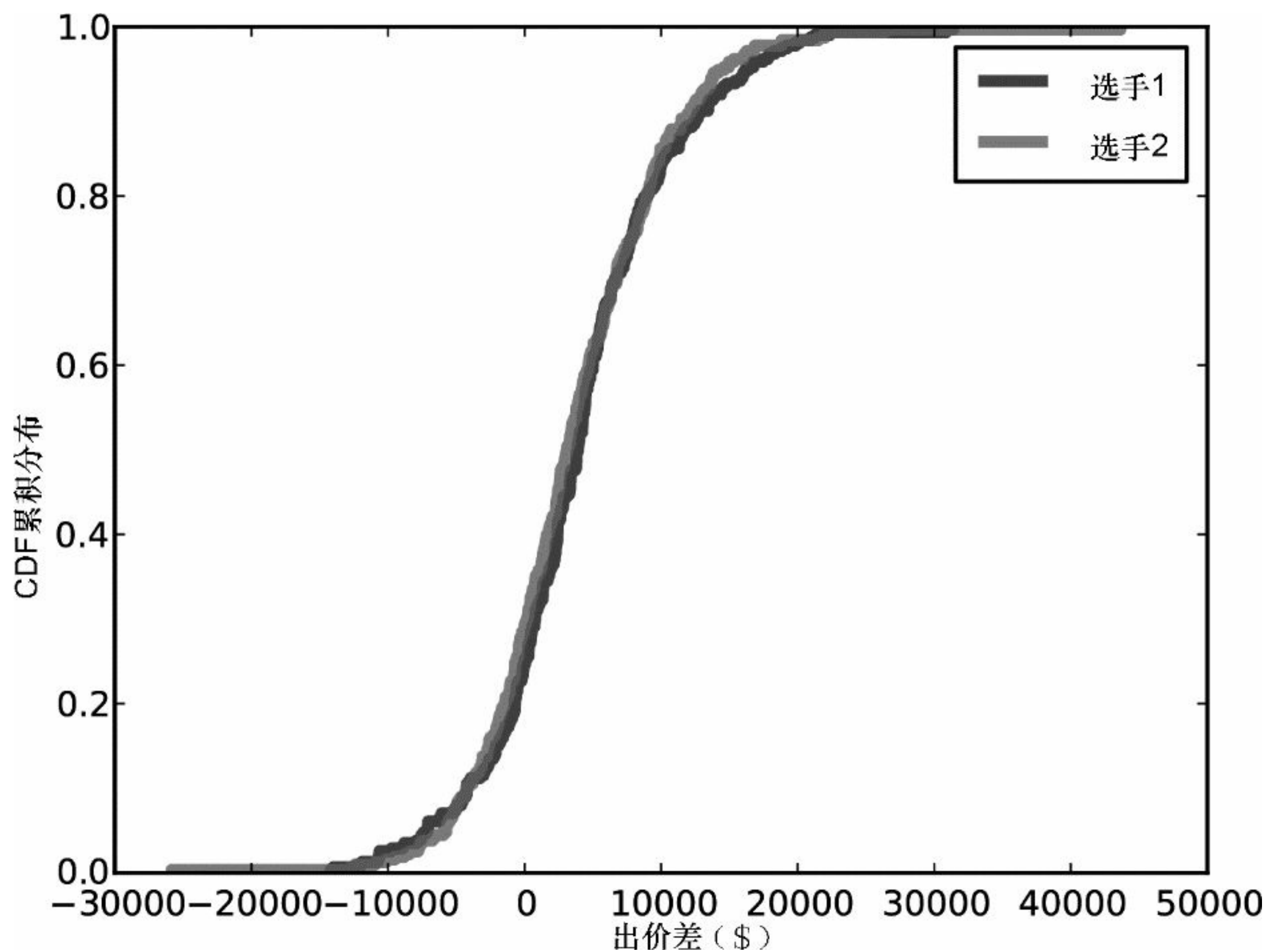


图6-2　选手出价和实际价格差，即出价差diff的累积分布（CDF）

出价差diff的定义是：

```
diff = price – bid （出价差＝奖品价格 – 选手出价）
```

当diff的值是负的，出价就太高。另一方面，我们可以使用这个分布计算参赛者出价过高的概率：第一组选手当中出高价的比例是25%；第二组选手当中出高价的比例是29%。

我们也可以看到出价是不对称的；即选手更倾向出低价。由于游戏规则这很好理解。

最后，我们可以使用这个分布估计参赛者猜测价格的可靠性（译注：由猜测误差衡量的）。这一步有点棘手，因为我们真不知道选手的猜测价格，只知道他们的出价。

因此我们必须做出一些假设。确切地说是认为error 的分布是一个和diff 方差相同，均值为0的高斯分布。

```
class Player(object):

    def __init__(self, prices, bids, diffs):
        self.pdf_price = thinkbayes.EstimatedPdf(prices)
        self.cdf_diff = thinkbayes.MakeCdfFromList(diffs)
        mu = 0
        sigma = numpy.std(diffs)
        self.pdf_error = thinkbayes.GaussianPdf(mu, sigma)
```

prices 是一个展品价格的序列，bids 是一个选手出价的序列，diffs 是一个出价差的序列，diff =price-bid 。

pdf_price 是一个价格的平滑PDF，由KDE给出。cdf_diff 是出价差的累积分布，如图6-2所示。pdf_error 是猜测误差分布的PDF，这里猜测误差=展品价格–猜测价格（error=price-guess ）。

再强调一下，我们使用diff 的方差估计error 的方差，这并不完美。因为选手的报价有时是有策略的。例如，如果选手2认为选手1出了高价，选手2可能会使用一个非常低的报价。在这种情况下出价差diff 就没有反映出猜测误差error 。如果发生较多这种情况，观测到的diff 方差会高估猜测误差error 的方差。然而，我认为这是一个合理的建模。

准备参赛的人可以通过看以前的比赛，记录下自己猜测价格和实际价格的猜测误差，从而评估自己猜测误差error 的分布。

6.6　似然度

现在我们准备好编写似然函数了。像通常那样，定义一个新类：

```
class Price(thinkbayes.Suite):
    def __init__(self, pmf, player):
        thinkbayes.Suite.__init__(self, pmf)
        self.player = player
```

pmf 代表了先验分布，player 是一个前面章节定义的Player对象。Likelihood 如下：

```
def Likelihood(self, data, hypo):
    price = hypo
    guess = data

    error = price - guess
    like = self.player.ErrorDensity(error)

    return like
```

hypo 是展示价格的假设。data是选手的最佳猜测价格。error 是他们的差，like 是给定假设后数据的可能性。

ErrorDensity 在Player 中定义：

```
# class Player:
    def ErrorDensity(self, error):
        return self.pdf_error.Density(error)
```

ErrorDensity 通过给定值的错误来评估pdf_error，结果是概率密度。因此它并不是真是一个概率。但请记住Likelihood 不需要计算概率；它只需要计算比例，只要所有likelihood 的比例系数是相同的，我们对后验分布进行归一化后就没问题了。

所以说概率密度是一个相当好的似然度方法。

6.7 更新

Player 提供了一个方法以选手的猜测来计算后验分布:

```
# class Player

    def MakeBeliefs(self, guess):
        pmf = self.PmfPrice()
        self.prior = Price(pmf, self)
        self.posterior = self.prior.Copy()
        self.posterior.Update(guess)
```

PmfPrice 生成PDF的离散近似价格，我们用其构建先验概率。

PmfPrice 使用MakePmf，评估pdf_price 序列的值：

```
# class Player

    n = 101
    price_xs = numpy.linspace(0, 75000, n)

    def PmfPrice(self):
        return self.pdf_price.MakePmf(self.price_xs)
```

为了得到后验概率，我们先复制先验概率，然后调用Update，Update 在每次假设中调用Likelihood 一次，用先验乘以似然度，最后归一化。

让我们回到最初的场景。假设你是选手1，当看到展品后你最佳的猜测是“奖品的总价格是20000美元”。

图6-3显示了实际价格的先验概率和后验概率。后验概率偏左些，因为你猜测的是先验概率的低值区间。

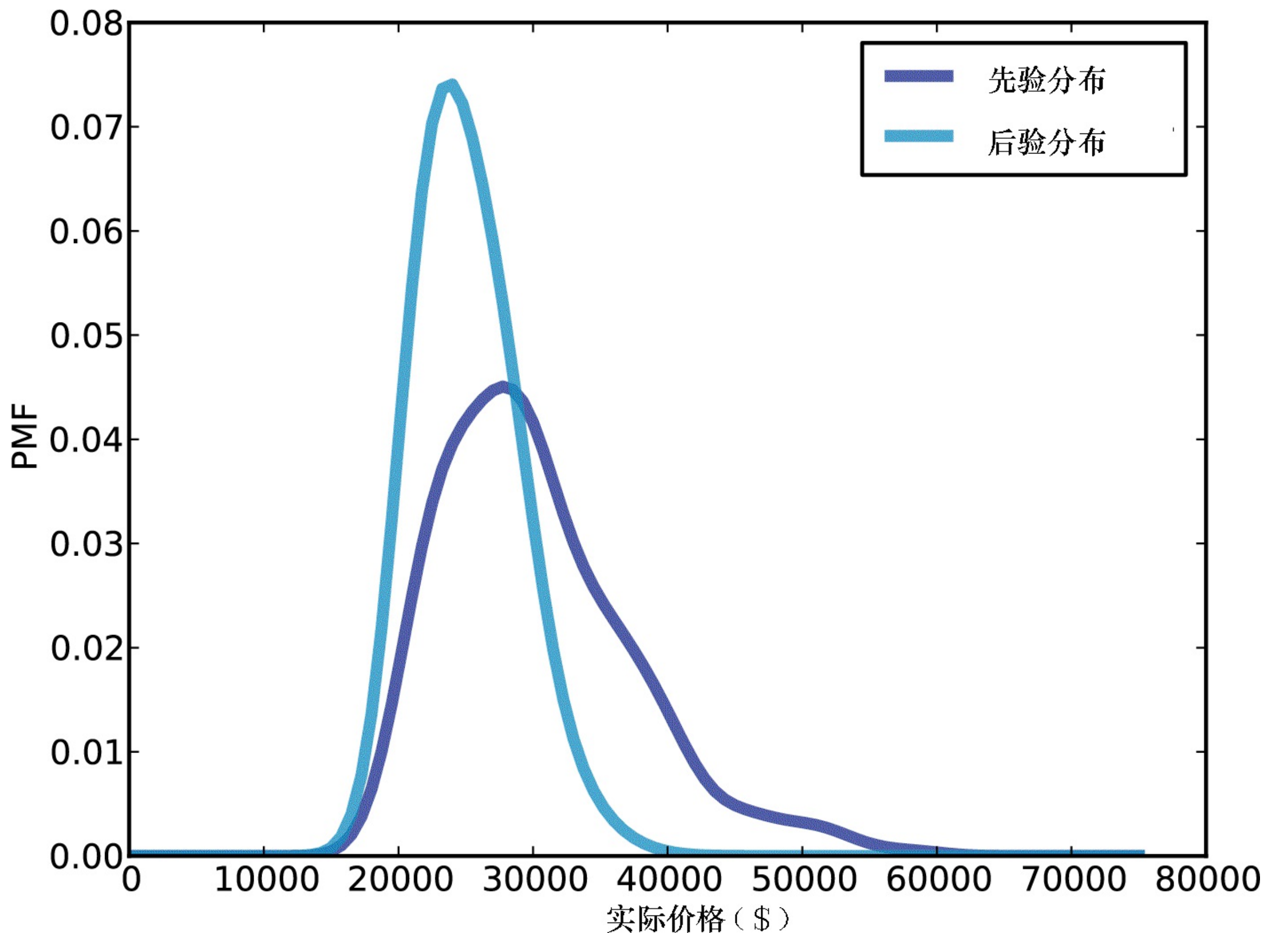


图6-3　基于一个20000美元价格的最佳猜测，选手1的前验和后验分布

在某种程度上这个结果是有意义的。在先验上最有可能的展品价格值是27750美元，你最好猜测是20000美元，而后验概率的均值在25096美元上下。

而从另一个角度上，你可能会发现这个结果的悖论，因为它表明如果你认为价格是20000美元，那么在考虑猜测误差的情况下，你应该相信价格是24000美元。

要想解决这个明显的矛盾，请记住你结合了两个信息来源，过去展品的历史数据和你看到奖品后作出的猜测。

我们把之前的历史数据当作先验概率和然后基于你的猜测去修正它。但同样的，我们也可以用你的猜测作为先验而把基于历史的数据进行修正（译注：修正和更新都是Update的意译，结合上下文使用）。

或者你可以这么思考——最有可能的展品价格并不是你最初的猜测值——于是这一点也就没那么奇怪了。

6.8 最优出价

现在我们有一个后验分布，我们可以使用它来计算最优报价，我定义为预期收益最大化的报价（见 *http://en.wikipedia.org/wiki/Expected_return* ）。

我将在本节中采用自顶向下的方法，这意味着我将先演示怎么使用，再演示为什么如此。如果你看到一个陌生的方法，不要担心，后面接着就是定义。

为计算最优报价，我写了一个类称为GainCalculator：

```
class GainCalculator(object):

    def __init__(self, player, opponent):
        self.player = player
        self.opponent = opponent
```

Player 和opponent 都是Player 对象。

GainCalculator 提供ExpectedGains，为每次出价计算投标序列和预期收益:

```
    def ExpectedGains(self, low=0, high=75000, n=101):
        bids = numpy.linspace(low, high, n)

        gains = [self.ExpectedGain(bid) for bid in bids]

        return bids, gains
```

low 和high 表示了出价可能的区间；n 是报价的次数。

ExpectedGains 调用ExpectedGain，计算对于一个给定的报价的预期值:

```
    def ExpectedGain(self, bid):
        suite = self.player.posterior
        total = 0
        for price, prob in sorted(suite.Items()):
            gain = self.Gain(bid, price)
            total += prob * gain
        return total
```

ExpectedGain 遍历后验概率的值，给定展品的实际价格后计算每次出价的回报。它针对概率进行加权计算然后返回总和。

ExpectedGain 调用Gain，Gain 通过报价和实际价格返回预期收益:

```
    def Gain(self, bid, price):
        if bid > price:
            return 0

        diff = price - bid
        prob = self.ProbWin(diff)

        if diff <= 250:
            return 2 * price * prob
        else:
            return price * prob
```

如果你出价高了将一无所获。反过来，我们计算出价和价格的差，这个决定了你获胜的概率。

如果差异小于250美元你就赢了。为简单起见，我假设展品有相同的价格。因为这个结果是罕见的，造成的差别不大。

最后，我们要基于diff 计算的赢的概率：

```
    def ProbWin(self, diff):
        prob = (self.opponent.ProbOverbid() +
                self.opponent.ProbWorseThan(diff))
        return prob
```

如果你的对手出价高，你赢。否则的话，你必须希望你的对手的出价差大于这个diff 值，Player 提供了一些方法来计算这两个可能性：

```
# class Player:

    def ProbOverbid(self):
        return self.cdf_diff.Prob(-1)
   def ProbWorseThan(self, diff):
       return 1 - self.cdf_diff.Prob(diff)
```

这段代码可能有些令人疑惑，因为这一计算过程是以对手的角度进行的，对手计算的正是“我出高价的可能性是多少?”和“我的出价差超过diff 的概率是多少?”

两种答案都是基于diff 的CDF。如果对手的差异小于或等于1，你赢。如果对手的diff 比你大，你赢。否则你输。

最后，计算最优报价的代码:

```
# class Player:

    def OptimalBid(self, guess, opponent):
        self.MakeBeliefs(guess)
        calc = GainCalculator(self, opponent)
        bids, gains = calc.ExpectedGains()
        gain, bid = max(zip(gains, bids))
        return bid, gain
```

给定一个对手和guess，OptimalBid 将计算出后验分布，实例化一个GainCalculator 计算各种投标可能区间的预期收益回报并返回最优报价和预期收益。太棒了!

图6-4显示了基于这样一个场景下的结果，选手1的最佳猜测是20000美元，选手2的猜测是40000美元。

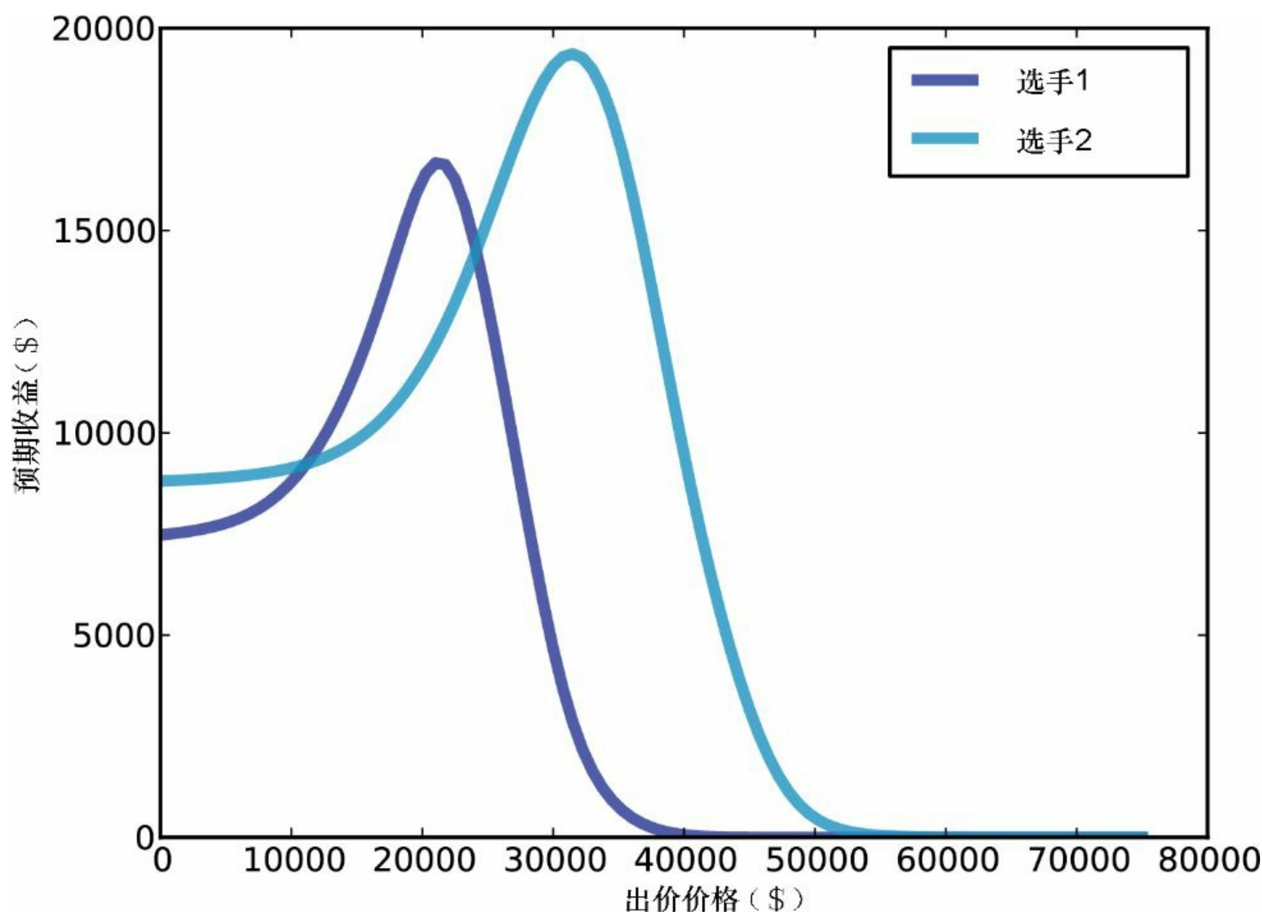


图6-4　当选手1的最佳猜测是20000美元，选手2的最佳猜测是40000美元时，预期收益的对照。

选手1的最优报价是21000美元，产生近16700美元的预期回报。

这是一个显得很不寻常的最优报价其实比选手的最佳猜测高一点的案例。

选手2的最优报价是31500美元，产生近19400美元的预期回报。这种最优报价小于最佳猜测的情况更加典型。

6.9　讨论

贝叶斯估计的特点之一就是结果来自后验分布这种形式。经典的估计通常会生成一个单一的点估计或置信区间，如果估计就是过程的最后一步，这是足够的。但如果你想以一个估计作为后续分析的输入，点估计和间隔往往没有多少帮助。

在这个例子中，我们使用后验分布来计算最优报价。给定出价的回报是不对称和不连续的（如果你高价，游戏失败），所以单纯分析很难解决这个问题。但用数值计算的方法就相对简单。

贝叶斯方法的初学者常常通过均值或者最大似然估计来描述后验分布，这种概述是有用的，不过如果你需要的仅仅就是这些内容，也许一开始就不必考虑贝叶斯方法。

在你需要将后验概率带入后续分析而进行模型决策时，贝叶斯方法就相当有用了，就如我们在本章做的一样。另外，进行预测时，贝叶斯方法也很有用，下一章我们会看到案例。

第7章　预测

7.1　波士顿棕熊队问题

在2010—2011的国家冰球联盟（NHL）总决赛中，我喜爱的7个赛季冠军波士顿棕熊队与我嗤之以鼻的温哥华加拿大人队对决。波士顿以0 : 1和2 : 3输掉了前两场比赛，以8 : 1和4 : 0赢了后续两场比赛。那么在赛季的这个时间点上，考虑波士顿赢下一场比赛的可能性是多少？赢得总冠军的概率是多少？

与往常一样，要回答这样的问题我们需要做一些假设。首先，有理由相信在冰球比赛中的进球得分至少近似于泊松过程，这意味着在比赛的任何时间上都有相同的得分可能。

其次，我们可以假设，长期来看对某一个特定的对手，每队都有一个单场平均得分数，记为λ 。

根据这些假设，我回答这个问题的策略是：

1．从以前的比赛统计资料，为λ 选择一个先验分布。

2．由前四场比赛的得分估计每队的λ 。

3．用λ 的后验分布来计算每队的进球分布，得分差的分布，还有每个球队下一场比赛获胜的概率。

4．计算每个队赢得赛季冠军的可能性。

要选择一个先验分布，我从 *http://www.nhl.com* 获得了一些统计数字，具体包括每队在2010—2011赛季平均每场进球的分布，大致是均值为2.8，标准差为0.3的高斯分布。

高斯分布是连续的，但我们会用离散的 Pmf 近似它，`thinkbayes` 提供了如下的`MakeGaussianPmf` 实现：

```
def MakeGaussianPmf(mu, sigma, num_sigmas, n=101):
```

```
    pmf = Pmf()
    low = mu - num_sigmas*sigma
    high = mu + num_sigmas*sigma

    for x in numpy.linspace(low, high, n):
        p = scipy.stats.norm.pdf(mu, sigma, x)
        pmf.Set(x, p)
    pmf.Normalize()
    return pmf
```

mu 和sigma 是高斯分布的均值和标准偏差。num_sigmas 是低于和高于Pmf均值的标准偏差数量，n 为Pmf中条目的个数。

同之前一样，我们使用numpy.linspace 构建一个在上下界区间（包括边界）上有着等间隔值的n 个条目的数组。

norm.pdf 估算出高斯概率密度函数（PDF）。

再回到比赛问题上，下面是一组有关λ 值的假设。

```
class Hockey(thinkbayes.Suite):

    def __init__(self):
        pmf = thinkbayes.MakeGaussianPmf(2.7, 0.3, 4)
        thinkbayes.Suite.__init__(self, pmf)
```

这样的先验分布是均值2.7，标准偏差0.3的高斯分布，范围是上下以均值为中心的4个sigma。

一如以往，我们必须决定如何表示每个假设，在这种情况下，我将$\lambda = x$ （x 是浮点值）作为假设。

7.2　泊松过程

在数理统计上，过程 是一个物理系统的随机模型（“随机”是指该模型包含某种随机量）。例如，一个伯努利过程是一个事件序列的模型，称为试验，其中每个试验有两种可能的结果，比如成功和失败。因此，对于一系列的硬币翻转事件，或者一系列的射击得分事件，伯努利过程

是一个天然的模型。泊松过程是伯努利过程的连续版本，一个事件在任何时间点上发生的概率是相同的。泊松过程可应用于到达商店的客户，公交车到站，或者冰球比赛的进球得分上。

在许多实际系统中，事件的概率随时间变化。客户在一天的某些时段去商店的可能性更高，公交车应当以固定的间隔到达，比赛中不同的时间段的得分有高低之分。

但所有模型都是为了简化事物，在这种情况下，以泊松过程模拟冰球比赛是一个合理的选择。豪雅，穆勒和鲁布纳（2010）分析德国足球联赛的得分，得出了相同的结论，见 *http://www.cimat.mx/Eventos/vpec10/img/poisson.pdf* 。

使用该模型的优点是，我们可以有效计算出每场比赛进球的分布以及进球时间点间隔的分布。具体地，如果一场比赛平均进球数是`lam`，每场比赛进球的分布可由泊松PMF给出：

```
def EvalPoissonPmf(lam, k):
    return (lam)**k * math.exp(-lam) / math.factorial(k)
```

得分间隔的分布由指数型PDF给出：

```
def EvalExponentialPdf(lam, x):
    return lam * math.exp(-lam * x)
```

我用了变量 `lam`，因为 `lambda` 是 Python 中的保留关键字。这两个函数都在`thinkbayes.py` 中。

7.3 后验

现在，我们可以计算一个给定`lam` 值的球队在一场比赛中进`k` 个球的可能性：

```
# class Hockey

    def Likelihood(self, data, hypo):
        lam = hypo
```

```
        k = data
        like = thinkbayes.EvalPoissonPmf(lam, k)
        return like
```

每个假设都是λ 的一个可能的值；数据是得分k ，利用这里的似然函数，我们可以为每个球队准备一个suite对象，接着用前四场比赛的得分进行修正。

```
    suite1 = Hockey('bruins')
    suite1.UpdateSet([0, 2, 8, 4])

    suite2 = Hockey('canucks')
    suite2.UpdateSet([1, 3, 1, 0])
```

图7-1显示了给定lam 的后验分布结果。根据前四场比赛的数据，最有可能的lam 值加拿大人队是2.6，棕熊队是2.9（译注：冰球比赛进一球得到一分，因此对“得分”和“进球”结合上下文灵活使用）。

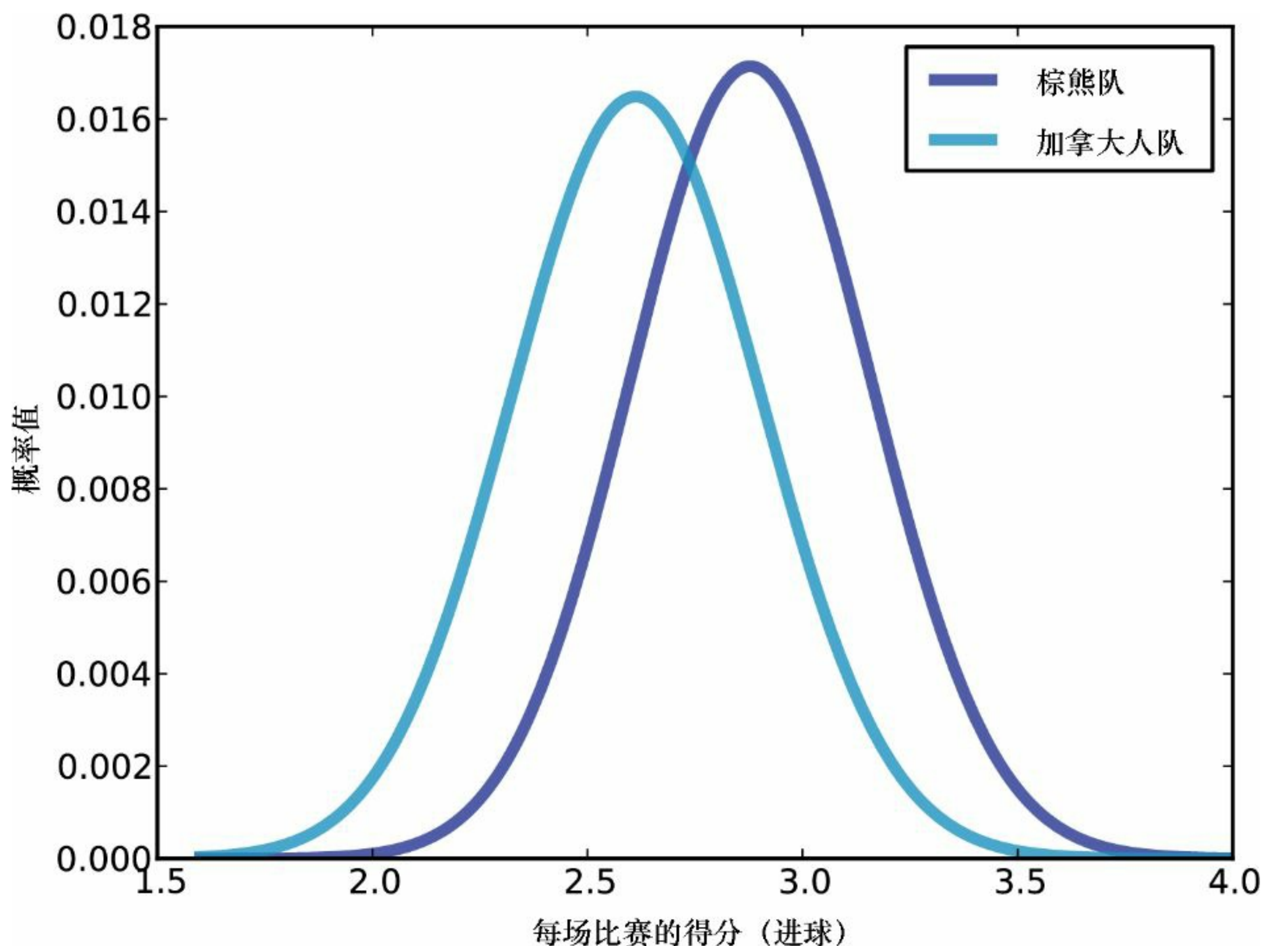


图7-1　每场比赛进球数的后验分布

7.4　进球分布

要计算每个团队赢得下一场比赛的概率，我们需要计算各球队得分的分布。

如果我们知道确切的`lam`取值，我们可以再次使用泊松分布。`thinkbayes`提供了一个方法计算泊松分布的分段近似值：

```
def MakePoissonPmf(lam, high):
    pmf = Pmf()
    for k in xrange(0, high+1):
        p = EvalPoissonPmf(lam, k)
        pmf.Set(k, p)
    pmf.Normalize()
    return pmf
```

```
```

所计算的Pmf值的范围从0到`high`。所以，如果`lam`值为3.4，我们可以计算如下：

```
lam = 3.4
goal_dist = thinkbayes.MakePoissonPmf(lam, 10)
```

我选择了上界10，因为比赛中得分超过10球的概率是相当低的。

目前为止这些都很简单，问题是我们不知道`lam`的确切值。不过还好，我们有`lam`可能值的分布。

对于`lam`的每一个值，得分是泊松分布。这样得分的整体分布就是这些泊松分布的混合分布，根据`lam`分布的概率进行加权。

给定`lam`的后验分布，下面是计算得分分布的代码：

```
def MakeGoalPmf(suite):
    metapmf = thinkbayes.Pmf()

    for lam, prob in suite.Items():
        pmf = thinkbayes.MakePoissonPmf(lam, 10)
        metapmf.Set(pmf, prob)

    mix = thinkbayes.MakeMixture(metapmf)
    return mix
```

对于`lam`的每一个值，我们创建一个泊松Pmf，并把它添加到元Pmf。我称其为元Pmf，

因为它是一个包含多个Pmfs值的Pmf对象。

然后我们用`MakeMixture`计算混合分布（参考`MakeMixture`的“混合分布”，第45页）。

图7-2 显示了由此产生的棕熊队和加拿大人队的进球分布。棕熊队在接下来的比赛中不太可能只进不到3球，更可能进4球以上。

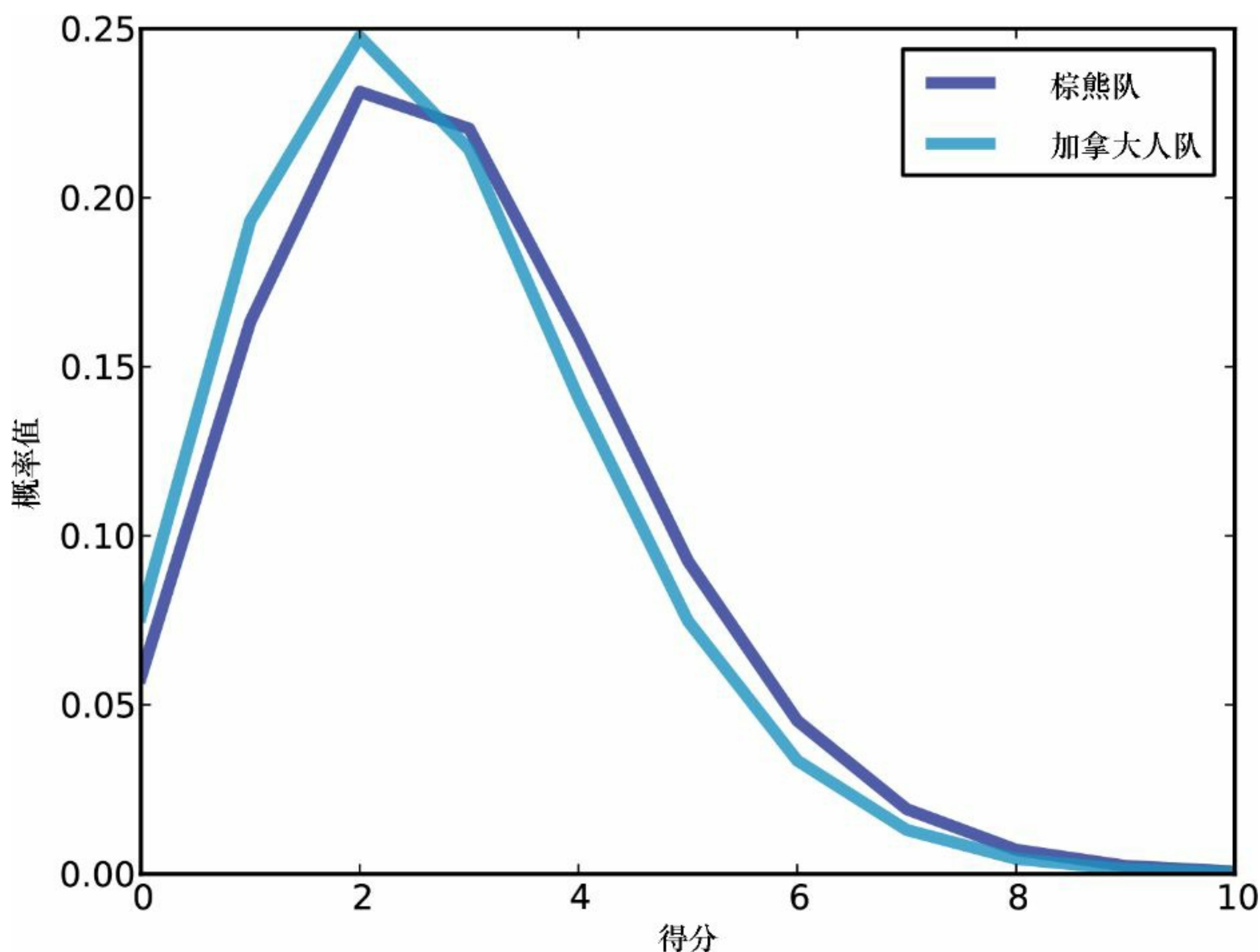


图7-2　单场比赛进球分布

7.5　获胜的概率

为了得到获胜的概率，首先我们计算两队进球得分差异的分布鉴别：

```
goal_dist1 = MakeGoalPmf (suite1)
goal_dist2 = MakeGoalPmf (suite2)
diff = goal_dist1 - goal_dist2
```

减法运算符调用`Pmf.__sub__`，它枚举一对值（两队得分）并计算差值。计算两个分布的差和相加操作一样，如我们在第40页“加数”看到的。

如果得分差别是正的，棕熊队赢；如果为负，加拿大人队赢；如果是0，就是平局：

```
    p_win = diff.ProbGreater (0)
    p_loss = diff.ProbLess (0)
    p_tie = diff.Prob (0)
```

由前一节中的得分的分布看，`p_win` 是46%，`p_loss` 是37%，`p_tie` 为17%。

在规定比赛时间结束时如果出现平局，两队会进行加时赛，直到其中一队得分。由于一旦进球比赛就会马上结束，这种加时赛形式被称为“突然死亡法则”。

7.6　突然死亡法则

为了计算在加时赛通过“突然死亡法则”赢球的概率，重要的统计指标不是每场比赛进球数，而是第一个进球的时间。假设的进球是一个泊松过程，意味着得分之间的时间服从指数分布。

给定`lam`，我们可以计算得分之间的时间：

```
lam= 3.4
time_dist = thinkbayes.MakeExponentialPmf (lam, high = 2 ,  n = 101)
```

`high` 是分布的上界。在这个例子中我选择2，因为超过2场比赛不进球的概率是很小的。`n` 是Pmf里面值的个数。

如果我们知道`lam` 的确切值，这就是全部需要的代码了。但我们不确定`lam`，我们知道的是`lam` 可能值的后验概率分布。

所以，如我们在计算得分分布中做的一样，我们建立一个元Pmf并且计算Pmfs的混合物。

```
def MakeGoalTimePmf(suite):
    metapmf = thinkbayes.Pmf()
```

```
    for lam, prob in suite.Items():
        pmf = thinkbayes.MakeExponentialPmf(lam, high=2, n=2001)
        metapmf.Set(pmf, prob)

    mix = thinkbayes.MakeMixture(metapmf)
    return mix
```

图7-3显示了生成的分布。对于时间小于1局（比赛的1/3，1场比赛有3局），棕熊队更容易得分。加时赛时间拖得越长，加拿大人队越可能得分。

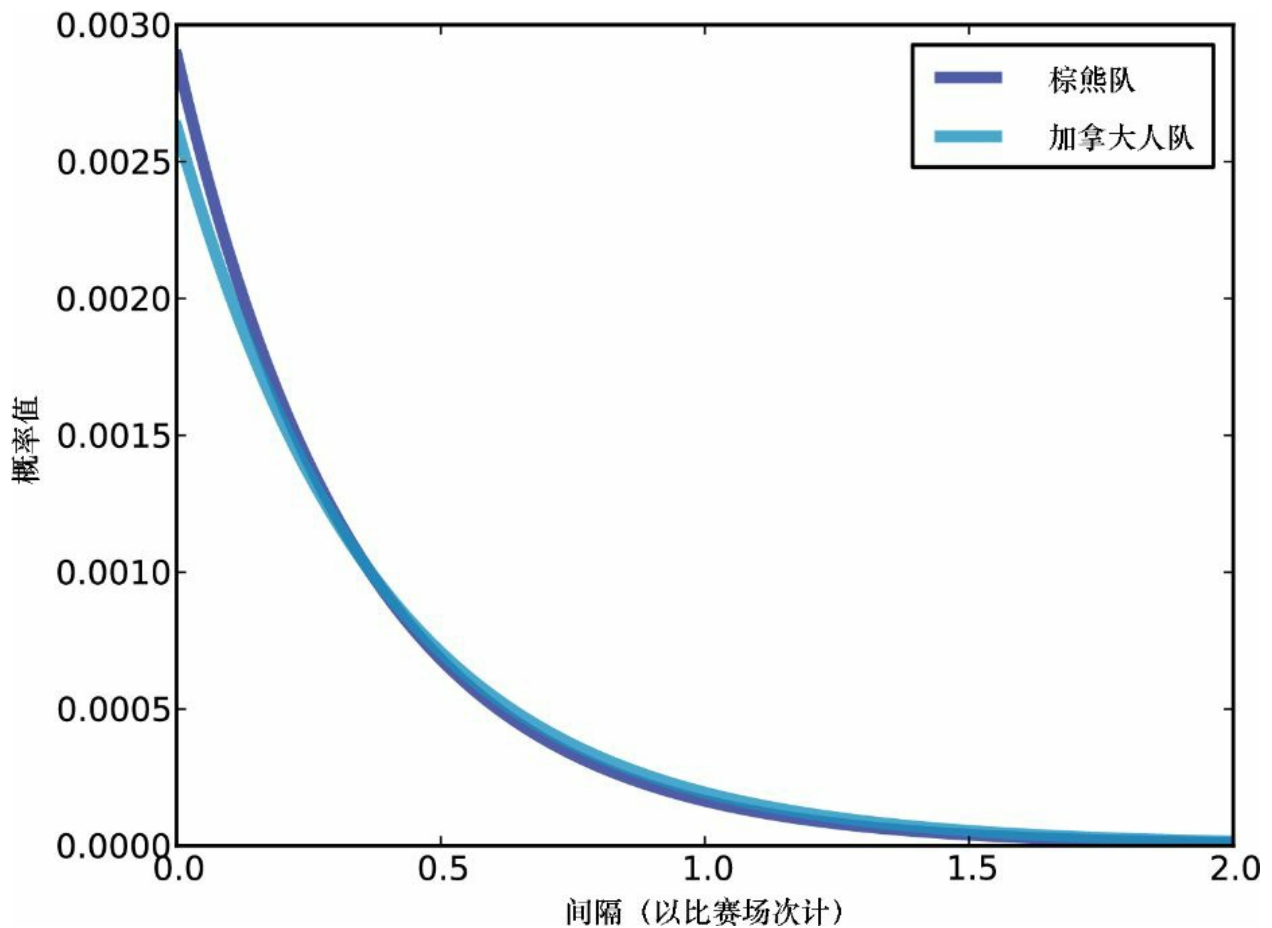


图7-3　两次得分间的间隔时间分布（时间以比赛场次计算）

我设置了相当高的n 值，以最大限度地减少平局的数量，因为两支球队是不可能同时得分的。

现在我们计算棕熊队首先得分的概率：

```
time_dist1 = MakeGoalTimePmf (suite1)
time_dist2 = MakeGoalTimePmf (suite2)
p_overtime = thinkbayes.PmfProbLess (time_dist1 ， time_dist2)
```

对棕熊队而言，赢得加时赛的概率为52%。

最后，获胜的总概率为常规时间赢得比赛的机会加上加时赛赢得比赛的概率。

```
p_tie = diff.Prob (0)
p_overtime = thinkbayes.PmfProbLess (time_dist1 ， time_dist2)

p_win = diff.ProbGreater (0)+ p_tie * p_overtime
```

所以对棕熊队而言，赢的下一场比赛的整体概率是55%。

如果要赢得整个赛季（总决赛）的话，棕熊队可以在下两场比赛中连胜或在接下来两场中输掉一场而赢得第三场。同样，我们可以计算出总的概率：

```
# win the next two
p_series = p_win**2

# split the next two, win the third
p_series += 2 * p_win * (1-p_win) * p_win
```

因此棕熊队赢得赛季总冠军的机会是57%。在2011年，他们确实做到了。

7.7　讨论

与往常一样，本章讨论的分析是基于模型决策的，建模几乎总是一个反复迭代的过程。

一般情况下，你要从一些简单的能产生近似答案的模型开始，找出错误的可能来源，并找到机会来改进模型。

在这个例子中，我会考虑以下选项：

- 我为每支队伍选择了一个基于平均得分的先验。但这个统计是所有对手的平均值。针对某一特定的对手，我们可以更灵活些。例如，如果最好锋线的队伍和最糟糕后防的队伍比赛，比赛得分预期目标可以是平均值以上的几个标准偏差。
- 关于数据，我只用了冠军赛前四场比赛的数据。如果两只球队在常规赛比赛过，我可以使用常规赛的结果数据。有些复杂的是球队成员在这期间的变动——由于受伤和赛季内的球员交易，所以最好给近期的比赛结果更多的权值。
- 为了充分利用所有可用的信息，我们可以从所有常规赛得分估计各队的进球率，用两只队伍间的比赛结果进行进一步的适度修正。这样做会有些复杂，但仍然是可行的。

对于第一种选择，我们可以用常规赛的比赛结果估计队伍间比赛的变化量。要感谢德克·霍格，我从 *http://forechecker.blogspot.com* ，上获得了常规赛每场比赛的进球得分数（不含加时赛）。

不同赛区的队伍在常规赛仅相遇1次到2次，所以我把重点放在进行4到6场比赛的队伍上。对于每一对球队，我计算他们每场比赛的平均进球作为λ 的估计值，然后绘制估计的分布。

这些估计的均值为2.8，不过标准差为0.85，比我们仅就每支队伍计算的更高。

如果我们采用这一较高的方差再次进行前验概率分析，棕熊队赢得系列赛的概率是80%，与采用较低方差得到的57%相比高很多。

所以，事实证明，这一结果对先验概率是敏感的，因此自然使我们思考，究竟采用多少数据进行分析才合理。

考虑到不同模型会产生的差异（高方差和低方差），这似乎说明值得我们投入一些精力来获得合适的前验概率。

本章的代码和数据都可以从 *http://thinkbayes.com/hockey.py* 和 *http://thinkbayes.com/hockey_data.csv* 获得。欲了解更多信息，请参阅前言的“代码指南”。

7.8 练习

练习7-1。

如果公交车到站间隔是20分钟，你到达公交站的时间是随机的，那么你等待公交车的时间从0到20分钟之间均匀分布。但在现实中，公交车抵达的间隔是有变动的。假设你正在等待一辆公交车，而你知道公交车抵达时间的历史分布。计算你等待时间的分布。

提示：假设公交车间隔为等概率的5分钟或10分钟。你刚好在一个10分钟发车间隔中到达公交站的概率是多少？

我在下一章提供了这个问题的一个解法。

练习7-2。

假设乘客到达公交车站是一个参数为 λ 的理想泊松过程。如果你到达车站，发现有 3人在等待，对你而言，距离上一趟公交车抵达过去了多久时间的后验分布是什么？

我下一章提供了这个问题的一种解法。

练习7-3。

假设你是一位在新的环境中进行害虫取样的生态学家。

你在测试地区安置了100个陷阱，第二天回去检查它们。你发现有37个陷阱被触发捕获害虫。一旦陷阱触发，它就不能再继续捕获其他昆虫直到被复位。

如果你重设陷阱，两天后（内）回来，你预期发现多少被触发的陷阱？计算这一后验/预测的分布。

练习7-4。

假设你是一个管理有100个灯泡公寓的大楼经理。你的责任是在灯泡破损时更换灯泡。

1月1日，所有100个灯泡都是好的。当你2月1日检查时，你发现有3个灯泡熄灭。假设你4月1日回来检查，你预期会有多少灯泡坏掉？

在前面的练习中，你可以合理地假定一个事件在任意时间发生的概率相同。对于灯泡，失效的可能性取决于灯泡的寿命。具体而言，旧灯泡的故障率会随着灯丝的蒸发而增加。

这个问题相比其他问题更开放，你将不得不作出决策模型。

你可能要了解一下Weibull分布（*http://en.wikipedia.org/wiki/Weibull_distribution*）。或寻找一些灯泡寿命曲线的相关信息。

第8章　观察者的偏差

8.1　红线问题

在马萨诸塞州，“红线”是连接剑桥和波士顿的地铁线路。我在剑桥工作的时候乘坐红线地铁从Kendall广场到南站，再转乘通勤铁路到Needham。上下班高峰期，红线列车平均每7～8分钟运行一趟。

当到达车站时，我可以根据站台上的乘客人数估算下一班车到达的时间。如果只有几个人，就推测刚刚错过了地铁，下一班地铁预计要等约7分钟。如果站台上有较多乘客，就估计地铁会很快到达。但是如果有相当多的乘客，则要怀疑列车未能如期运行，所以会回到街上叫出租车出行。

在等待火车时，我思考了怎样通过贝叶斯估计帮助我预测等待时间，并决定什么时候应该放弃坐火车而改为乘坐出租车。本章将介绍我采用的分析过程。

本章内容源自Brendan Ritter和Kai Austin负责的一个项目，他们和我在欧林学院同教一个班。本章中的代码可以从*http://thinkbayes.com/redline.py* 得到。我用来收集数据的代码在*http://thinkbayes.com/redline_data.py* 。欲了解更多信息，请参见前言的“代码指南”。

8.2　模型

在分析前，我们必须决定一些建模细节。首先，我将旅客抵达车站当作泊松过程，这意味着我假设乘客可能在任何时间等概率到达，乘客有一个未知的到达率λ，以每分钟到达的乘客计量。因为我在很短的时间段内观察乘客，而且是在每天的同一时间，所以我假设λ为常数。

另一方面，列车的到达过程不是泊松的。高峰时间从终点（灰西鲱站）去波士顿的列车每隔7～8分钟发出，但到Kendall广场的时候，列车间隔在3～12分钟内变化。

为了收集关于列车发车间隔的数据，我编写了下载实时数据的脚本*http://www.mbta.com/rider_tools/developers/*，选择往南到达Kendall广场的列车，并在数据库中记录其到达时间。脚本在每个工作日下午4点到下午6点运行，持续5天，每天记录了 15次列车到达。然后我计算前后到达列车的时间间隔，这些分布的差别如图8-1 所示，标为z 。

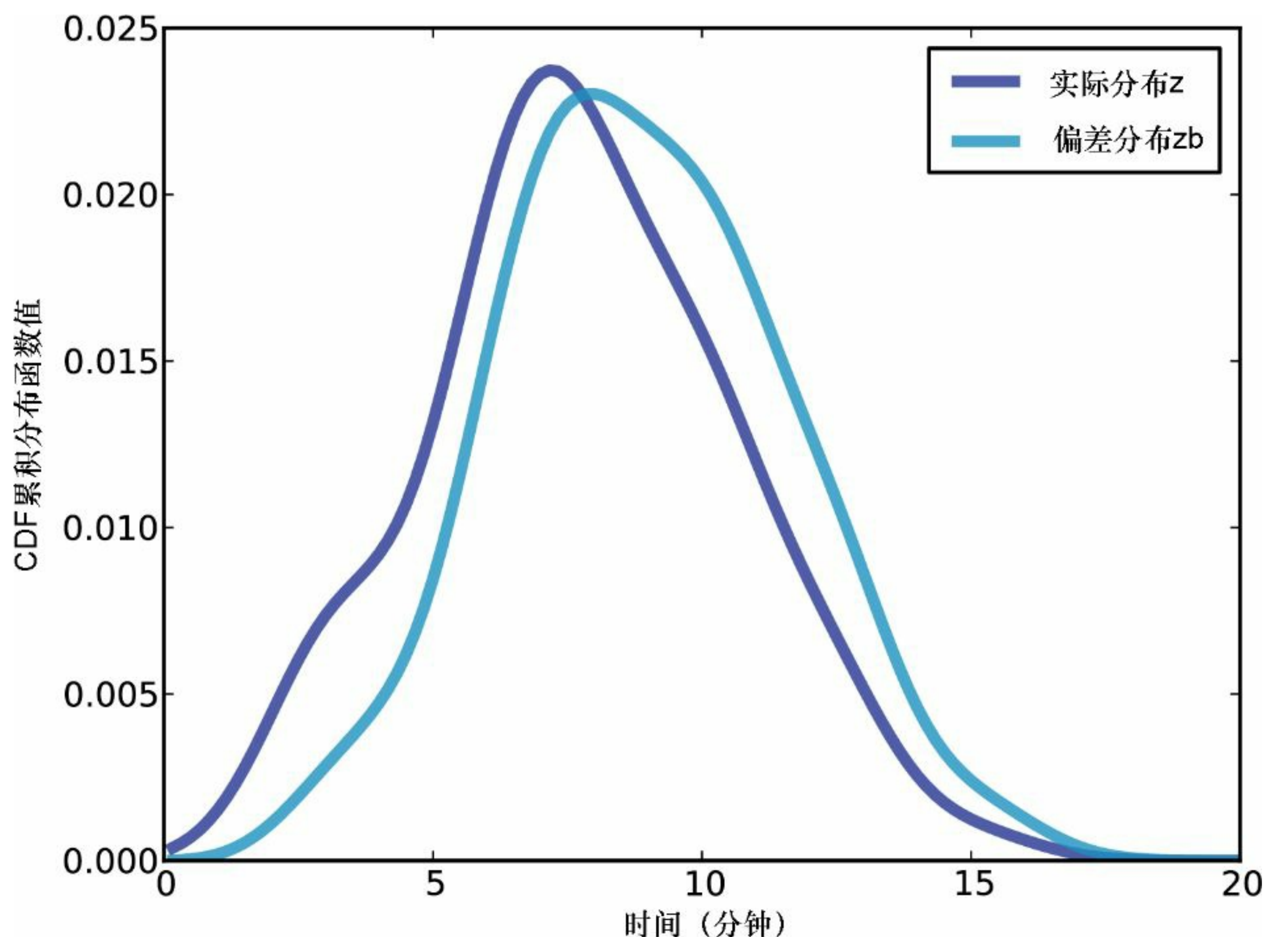


图8-1　根据收集到的数据绘制的列车间隔的PMF，以KDE平滑处理（z为实际分布；zb是由乘客看到列车间隔的偏差分布）

如果你下午4点到下午6点在站台记录列车的间隔，这就是你看到列车间隔时间的分布。但是如果你随机到达站台（不管列车时刻表），会看到一个与此不同的分布，随机到达的乘客所看到的列车间隔的平均值，比实际的平均值要高一些。

为什么？因为乘客到达的时间间隔更可能是一个较大的区间。考虑一个简单的例子：假设列车间隔是5分钟或者10分钟（相等的概率）。在这种情况下，列车之间的平均时间是7.5分钟。

但乘客更可能在10分钟的时段内到达而不是在5分钟内，事实上前者是后者的两倍。如果我们调查到站旅客会发现，其中2/3在10分钟的时段内到达，5分钟时段内到达的只有1/3。所以到站乘客观察到的列车间隔平均值是8.33分钟。

这种观察者偏差 在许多情况下出现。学生们认为班级比实际的要大是因为他们经常上大课，飞机上的乘客认为飞机比实际更满是因为他们常常乘坐满员的航班。

在每种情况下，实际分布中的值都按照比例被过采样了。例如，在红线上，差距就是两倍大。

所以，有了列车间隔的实际分布，我们可以计算得到乘客看到的列车间隔分布。`BiasPmf` 进行这个计算：

```
def BiasPmf(pmf):
    new_pmf = pmf.Copy()

    for x, p in pmf.Items():
        new_pmf.Mult(x, x)

    new_pmf.Normalize()
    return new_pmf
```

`pmf` 是实际的分布；`new_pmf` 是偏分布。在循环中，我们将每个值的概率`x` 乘以观测到的似然度，其正比于`x`，然后我们对结果归一化。

8.3 等待时间

等待时间称之为`y`，是乘客到达时刻和下一趟列车到达时刻之间的时间。经过时间称之为`x`，是乘客到达时刻和上一趟列车到达时刻之间的时间。这样定义使得`zb = x +y` 。

给定`zb` 的分布，我们可以计算出`y` 的分布。我先从一个简单的情况开始，然后再一般化。假设如前面的例子，`zb` 为5分钟的概率是1/3，10分钟的概率就是2/3。

如果我们在5分钟间隔内随机到达，`y` 均匀分布于0至5分钟内。如

果我们在10分钟的间隔到达，y 均匀分布于0到10分钟内。所以整体分布是根据每一个间隔的概率加权了的均匀分布的混合分布。

下面的函数将计算zb 的分布和y 的分布：

```
def PmfOfWaitTime(pmf_zb):
    metapmf = thinkbayes.Pmf()
    for gap, prob in pmf_zb.Items():
        uniform = MakeUniformPmf(0, gap)
        metapmf.Set(uniform, prob)

    pmf_y = thinkbayes.MakeMixture(metapmf)
    return pmf_y
```

PmfOfWaitTime 通过映射每个均匀分布和其概率来构建一个元Pmf。然后，它使用45页“混合分布”中的MakeMixture ，计算混合分布。

PmfOfWaitTime 还使用了MakeUniformPmf ，定义为：

```
def MakeUniformPmf(low, high):
    pmf = thinkbayes.Pmf()
    for x in MakeRange(low=low, high=high):
        pmf.Set(x, 1)
    pmf.Normalize()
    return pmf
```

low 和high 决定了均匀分布的范围（含两端）。最后，MakeUniformPmf 使用了MakeRange ，此处定义为：

```
def MakeRange(low, high, skip=10):
    return range(low, high+skip, skip)
```

MakeRange 定义了一组等待时间（以秒表示）的可能值。默认情况下，它将范围划分为10秒的时间间隔。

为了封装这些分布的计算过程，我创建了一个类WaitTimeCalculator ：

```
class WaitTimeCalculator(object):

    def __init__(self, pmf_z):
        self.pmf_z = pmf_z
        self.pmf_zb = BiasPmf(pmf)

        self.pmf_y = self.PmfOfWaitTime(self.pmf_zb)
        self.pmf_x = self.pmf_y
```

参数pmf_z 是z 的非偏差分布。pmf_zb 是乘客看到的列车间隔的偏差分布。pmf_y 是等待时间的分布。pmf_x 是经过的时间的分布，它和等待时间分布是一样的。想知道为什么？记得对于一个zp 的一个特定值，y 的分布是从0到zp 均匀的，再考虑到x = zp – y，因此x 的分布也是从0到zp 均匀的。

图8-2显示了z 、zb 和y 的分布——基于我从Red Line网站上收集的数据。

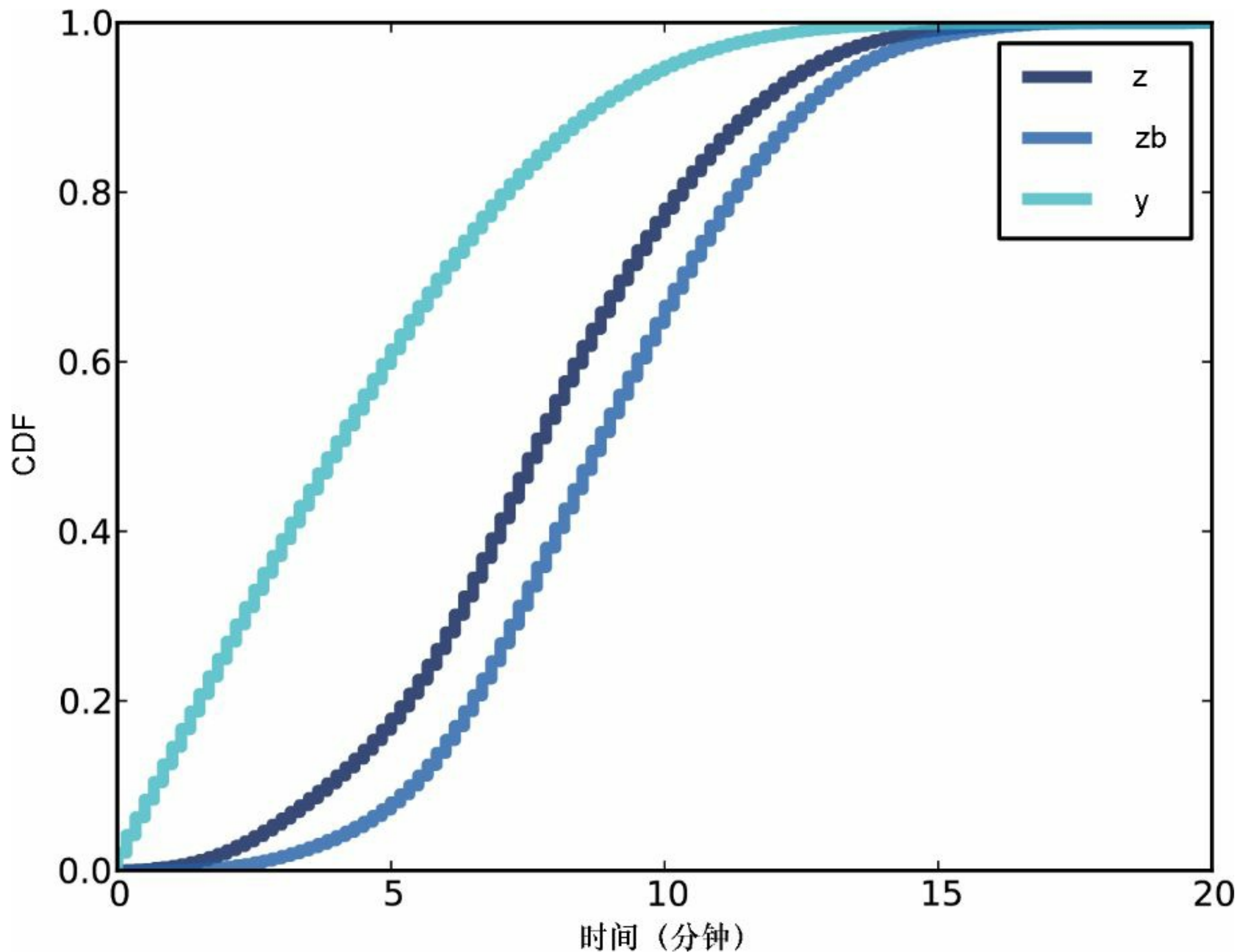


图8-2　z，zb，乘客等待时间y的CDF

为了解释这些分布，我从Pmfs切换到Cdfs。大多数人都更熟悉Pmfs，但我认为一旦你习惯了，Cdfs更容易解释。如果要在同一坐标轴上绘制多个分布，就应该用Cdfs。

z 的平均值为7.8分钟。zb 的平均值为8.8分钟，高出z 约13%。y均值为4.4分钟，是zb 均值的一半。

顺便提一下，根据红线调度站的报告显示，在高峰期间列车运行间隔是9分钟/次。这接近zb 的平均值，但比z的平均值高。通过与MBTA铁路局代表的邮件沟通，报告中的列车间隔时间是有所保留的，以提供一些回旋余地。

8.4　预测等待时间

让我们回到初始问题：设想一下，当我到达站台时看到有10人在等待。在下一班车到达前，我预期要等待多长时间呢？

与往常一样，让我们从这个问题的最简单的版本开始，然后找到最终答案。假设我们给出z的实际分布，而我们知道乘客到达率λ是每分钟2名乘客。

在这种情况下，我们可以：

1．用z 的分布来计算zp 的先验分布，乘客所看到的列车间隔分布。

2．然后，我们可以使用乘客数量来估计x 的分布，即上一趟火车离开后经过的时间。

3．最后，我们使用关系y = zp - x 可得y 的分布。

第一步是创建一个WaitTimeCalculator，封装zp，x 和y 的分布——在考虑乘客的数目之前。

```
    wtc = WaitTimeCalculator(pmf_z)
```

pmf_z 是给定的间隔时间的分布。

接下来的步骤是创建一个ElapsedTimeEstimator，它封装了x 的后验分布和y 的预测分布。

```
    ete = ElapsedTimeEstimator (wtc,
                                lam= 2.0/60,
                                num_passengers = 15)
```

参数是WaitTimeCalculator，乘客到达率lam（表示为乘客人数/秒）和站台上看到的乘客数量（假设是15）。

ElapsedTimeEstimator 的定义：

```
class ElapsedTimeEstimator(object):
```

```
    def __init__(self, wtc, lam, num_passengers):
        self.prior_x = Elapsed(wtc.pmf_x)

        self.post_x = self.prior_x.Copy()
        self.post_x.Update((lam, num_passengers))

        self.pmf_y = PredictWaitTime(wtc.pmf_zb, self.post_x)
```

prior_x 和posterior_x 是经过时间的先验和后验分布。pmf_y 是等待时间的预测分布。

ElapsedTimeEstimator 使用Elapsed 和PredictWaitTime，定义如下。

Elapsed 是表示x 的假想分布的Suite对象。x的先验分布直接由WaitTimeCalculator 得到。然后，我们使用这些数据，包括到达率，lam 和站台上乘客的数量计算后验分布。

下面是Elapsed 的定义：

```
class Elapsed(thinkbayes.Suite):

    def Likelihood(self, data, hypo):
        x = hypo
        lam, k = data
        like = thinkbayes.EvalPoissonPmf(lam * x, k)
        return like
```

与往常一样，Likelihood 接受一个假设和数据，并计算该假设下数据的似然度。在这个例子里面hypo 是上一趟列车后经过的时间，data 是一个包括lam 和乘客数量的元组。

数据的似然度是给定到达率lam 下，x 时间内k 次列车抵达的概率。我们利用一个泊松分布的PMF 来计算它。

最后，PredictWaitTime 的定义是：

```
def PredictWaitTime(pmf_zb, pmf_x):
```

```
    pmf_y = pmf_zb - pmf_x
    RemoveNegatives(pmf_y)
    return pmf_y
```

pmf_zb 是列车间隔的分布情况；pmf_x 是经过时间的分布（根据对乘客数量的观察得到）。由于y = zb - x，我们可以计算：

```
    pmf_y = pmf_zb - pmf_x
```

减法运算符调用Pmf.__sub__，其中列举了所有zb 和x 对，计算其差，将结果加总到pmf_y 。

由此产生的Pmf包括一些显然不可能的负值。例如，如果你是在5分钟的间隔期间到达的，你的等待时间不可能超过5分钟。RemoveNegatives 会移除这些不可能的值并重新归一化。

```
def RemoveNegatives(pmf):
    for val in pmf.Values():
        if val < 0:
            pmf.Remove(val)
    pmf.Normalize()
```

图8-3显示了结果。x 的先验分布和y 一样。x 的后验分布表明，看到站台上的15名乘客后，考虑到自上一趟车过后的时间大概是5～10分钟，所以我们预计下一班列车会在5分钟内到达，置信度为80%。

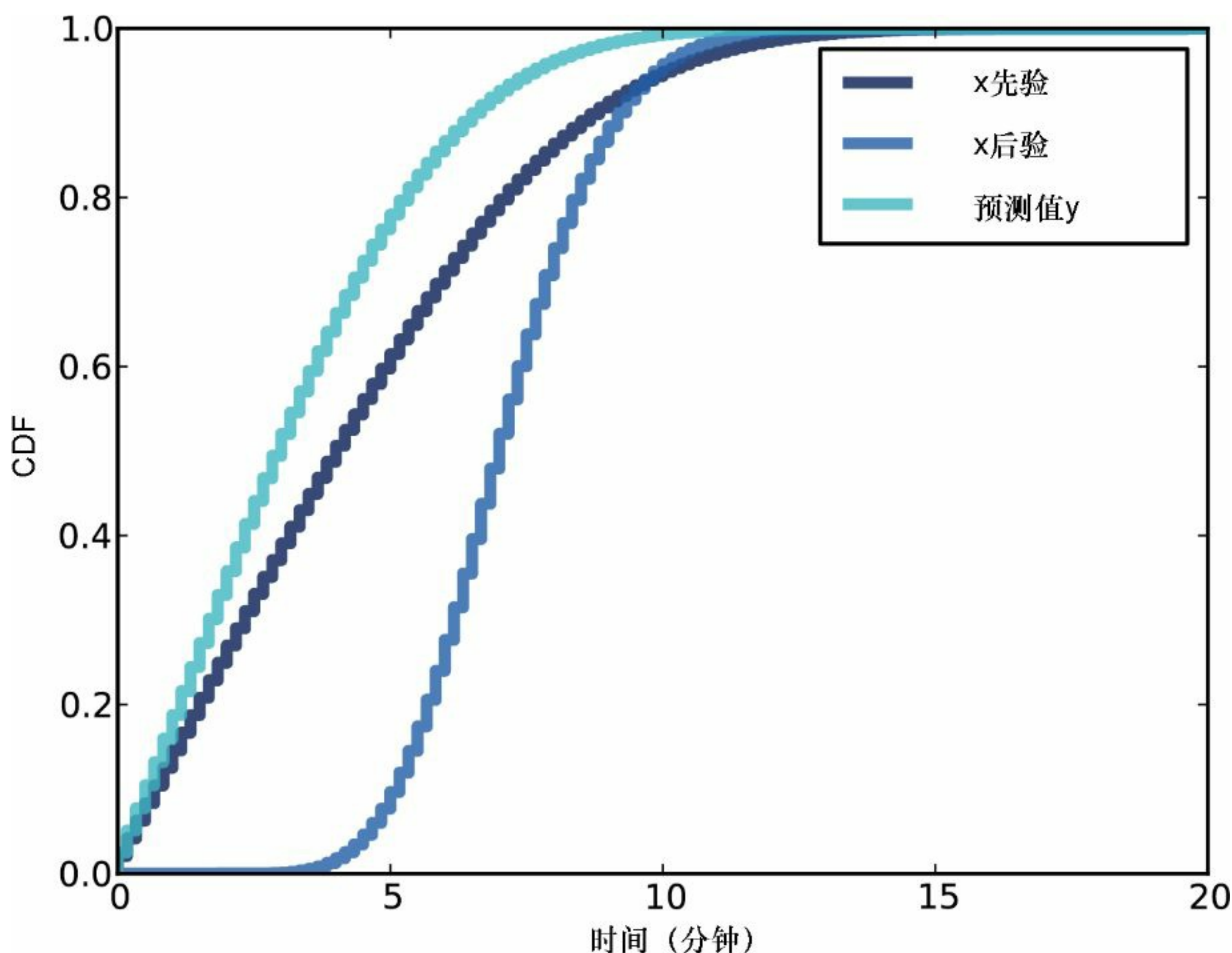


图8-3　x的先验分布和后验分布，以及预测的y值

8.5　估计到达率

到目前为止的分析基于我们已知（1）列车间隔的分布（2）乘客到达率的假设。现在，我们已经准备好开始处理第二个假设。

假设你刚搬到波士顿，所以你不了解红线地铁的乘客到达率。利用几天上下班时间，就可以做至少是可量化的猜测。只要再花一点心思，你甚至可以定量的估计λ。每一天你到达站台时，你应该注意时间和到达乘客的数量（如果站台太大，你可以选择一个样本区域）。然后记录自己的等待时间，以及在你等待期间新到站的乘客数量。

5天后，你可能得到这样的数据：

```
k1      y      k2
--     ---     --
17     4.6      9
22     1.0      0
23     1.4      4
18     5.4     12
4      5.8     11
```

其中k1 是当你到达时，正在等候的乘客数，y 是你的等待时间，k2 为等待期间到达的乘客数量。

一个多星期的记录中，你等待时间是18分钟，看到36名乘客到达，因此可以估计，到达率是每分钟2名乘客。就实验来说，这一估计足够了，但为了完整起见，我会计算λ 的后验分布，然后演示怎么样在后面的分析中利用该分布。

ArrivalRate 是个代表λ 假设的Suite 对象。与往常一样，Likelihood 接收假设和数据，计算出假设下的数据似然度。

在例子里面，假设是λ 的取值。数据是y 、k 数据对，其中y 是一个等待时间，k 是到达的乘客人数。

```
class ArrivalRate(thinkbayes.Suite):

    def Likelihood(self, data, hypo):
        lam = hypo
        y, k = data
        like = thinkbayes.EvalPoissonPmf(lam * y, k)
        return like
```

这一Likelihood 看起来很熟悉，它和第75页“预测等待时间”里的Elapsed.Likelihood 几乎一模一样。区别在于Elapsed.Likelihood 里假设是经过的时间x ，在ArrivalRate.Likelihood 里假设是lam 到达率。但两个例子里面，似然度都是在已知lam 的条件下，一段时间里遇到k 个到达（乘客）的可能性。

ArrivalRateEstimator 封装估算λ 的过程。参数passenger_data ，是一个包括k1 ，y ，k2 元素的元组，具体数据如前文所示。

```
class ArrivalRateEstimator(object):

    def __init__(self, passenger_data):
        low, high = 0, 5
        n = 51
        hypos = numpy.linspace(low, high, n) / 60

        self.prior_lam = ArrivalRate(hypos)
        self.post_lam = self.prior_lam.Copy()
        for k1, y, k2 in passenger_data:
            self.post_lam.Update((y, k2))
```

`__init__` 构建假设，这是`lam`假设值的序列，然后生成先验分布`prior_lam`。`for`循环以数据更新前验概率，产生后验分布`post_lam`。

图8-4给出了先验和后验分布。正如预期的那样，均值和中位值都在观察得到的值附近，每分钟2名乘客。但我们不确定后验分布的范围是否是由于λ基于小样本的原因。

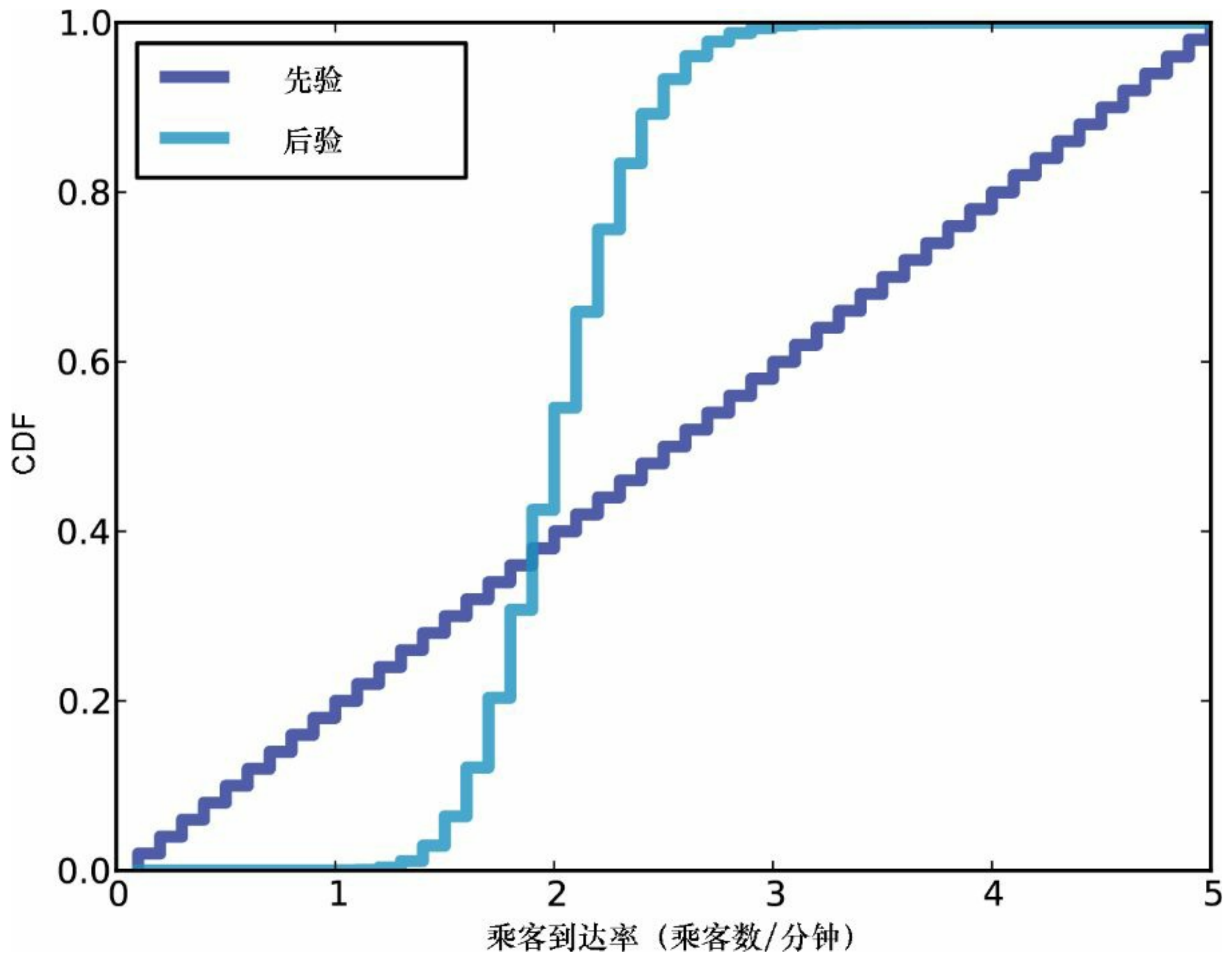


图8-4　基于5天乘客数据的lam的前验和后验分布

8.6　消除不确定性

无论何时，分析中总有一些输入量带来的不确定性，我们可以通过下面这个步骤将这一因素考虑进来：

1．实现基于不确定参数的确定值分析（在本例中是λ）。

2．计算不确定参数的分布。

3．对参数的每个值进行分析，并生成一组预测分布。

4．使用参数分布所对应的权值计算出预测分布的混合分布。

我们已经完成了步骤1和步骤2。我写了一个类`WaitMixtureEstimator`处理步骤3和步骤4。

```
class WaitMixtureEstimator(object):

    def __init__(self, wtc, are, num_passengers=15):
        self.metapmf = thinkbayes.Pmf()

        for lam, prob in sorted(are.post_lam.Items()):
            ete = ElapsedTimeEstimator(wtc, lam, num_passengers)
            self.metapmf.Set(ete.pmf_y, prob)

        self.mixture = thinkbayes.MakeMixture(self.metapmf)
```

`wtc`是包含`zb`分布的`WaitTimeCalculator`实例。`are`则是包含了`lam`分布的`ArrivalTimeEstimator`实例。第一行创建了一个元Pmf来映射`y`的可能分布和其概率。对于`lam`的每一个值，我们用`ElapsedTimeEstimator`计算`y`的相应分布，并将其存储在元Pmf。然后我们用`MakeMixture`来计算混合分布。

图8-5显示了结果。背景中的阴影线表示了`y`对应于`lam`每个值的分布，细线表示似然度。粗线是这些分布的混合分布。

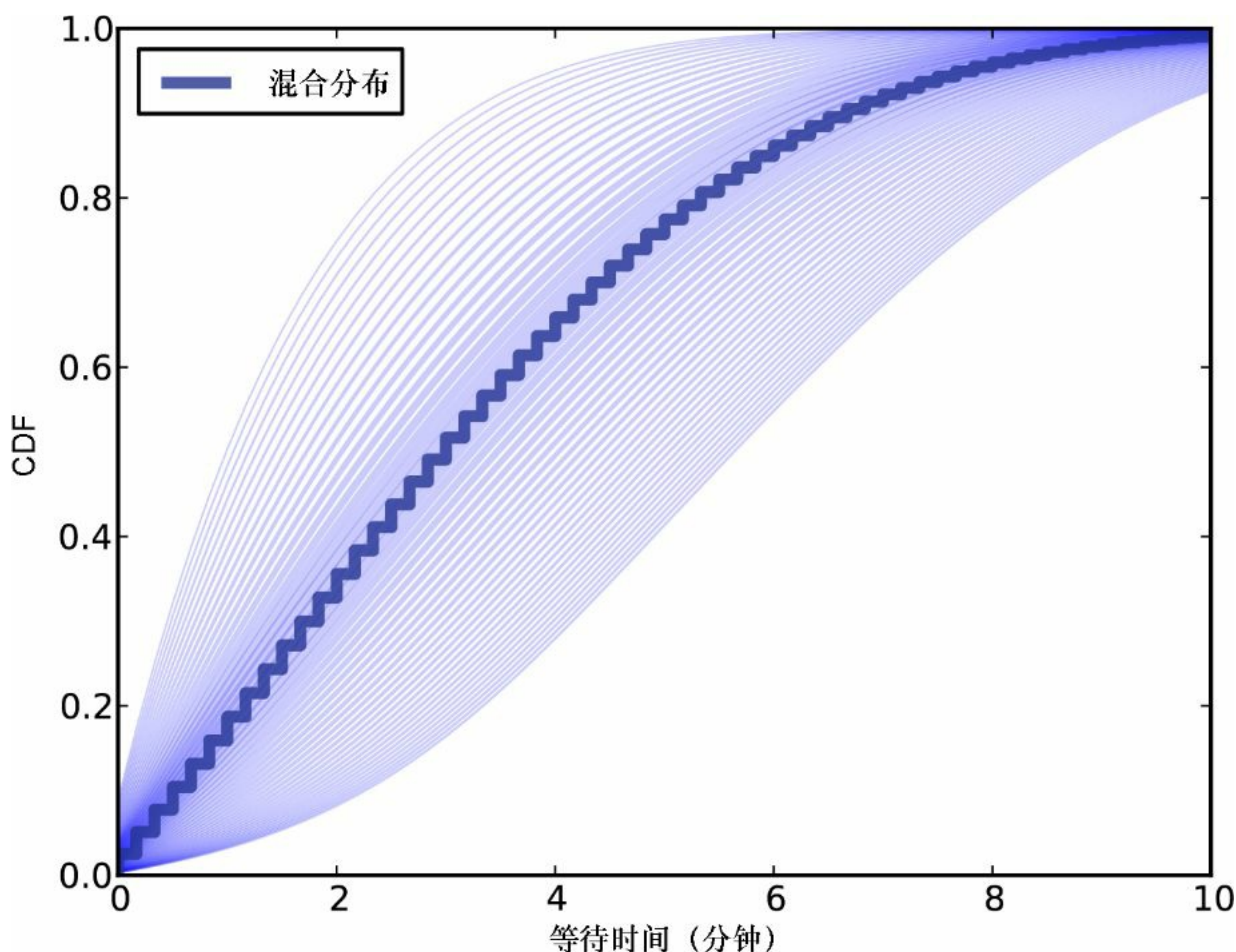


图8-5　对应了lam所有可能值的y的预测分布

在这种情况下，我们可以用`lam`的单点估计得到一个非常类似的结果。因此就实用而言，将估计的不确定性包含进来不是必需的。

在一般情况下，如果系统响应是非线性的，那么包括可变性就很重要了。此时，输入的微小变化都会引起输出的较大变化，而本例中，`lam`的后验变化很小，对于小的扰动，系统的响应近似线性。

8.7　决策分析

现在，我们可以使用站台上乘客人数预测等待时间的分布了。让我们处理问题的第二部分：我应该在何时放弃等待列车去叫一辆出租车？

请记住，在初始的场景中，我会去南站乘坐通勤铁路。假设我下班足够早，所以足以等待15分钟在南站乘车。

在这种情况下，我想知道的是“y 超过15分钟”作为num_passengers 的函数的概率。用“预测等待时间”里的分析方法这很容易。在num_passengers 的区间上运行这个分析。

但有一个问题。该分析对长时间延误的频次敏感，而由于长时间延误罕见，因此很难估计其时间延误发生频次。

我只有一周的数据，观察到的最长延误是15分钟。所以我无法准确估计长时间延误的频次。不过我还是可以使用以前的观察来进行至少是粗略的估计。

在一年时间乘坐红线的过程中，我看到了由于信号问题、停电、其他车站的警察行动造成的3个长时间延误，所以我估计大约每年有3次长时间延误。

但请记住我的看法是偏颇的。我更倾向于观察长时间延误是因为它们影响了大批乘客。所以，我们应该把我的意见作为zb 的样本，而不是z 的。下面是我们怎样做到这一点。

在乘坐地铁通勤那一年，我乘坐红线约220次。所以我用观察到的间隔时间gap_times 产生了220个列车间隔的样本，并计算它们的Pmf：

```
n = 220
cdf_z = thinkbayes.MakeCdfFromList (gap_times)
sample_z = cdf_z.Sample (n)
pmf_z = thinkbayes.MakePmfFromList (sample_z)
```

接下来，我偏置pmf_z 得到zb 的分布情况，抽取样本，然后添加了30分钟、40分钟和50分钟的三次延误（以秒表示）：

```
cdf_zp = BiasPmf (pmf_z ). MakeCdf()
sample_zb = cdf_zp.Sample(n)+ [ 1800 , 2400 ,  3000]
```

Cdf.Sample 比Pmf.Sample 更高效，因而一般会更快地将Pmf转换成Cdf。

接下来，我以zb 的样本用KDE来估计Pdf，然后将Pdf转换为Pmf：

```
    pdf_zb = thinkbayes.EstimatedPdf (sample_zb)
    xs = MakeRange(low= 60)
    pmf_zb = pdf_zb.MakePmf (xs)
```

最后，我反偏置zb 的分布来获得z 的分布，用z 创建WaitTimeCalculator：

```
    pmf_z = UnbiasPmf (pmf_zb)
    wtc = WaitTimeCalculator (pmf_z)
```

这个过程是复杂的，但所有的步骤都是我们所见过的操作。现在我们准备进行计算一个长时间等待的概率。

```
def ProbLongWait(num_passengers, minutes):
    ete = ElapsedTimeEstimator(wtc, lam, num_passengers)
    cdf_y = ete.pmf_y.MakeCdf()
    prob = 1 - cdf_y.Prob(minutes * 60)
```

根据平台上的乘客人数，ProbLongWait 用ElapsedTimeEstimator 提取等待时间的分布，并计算等待时间超过minutes 的概率。

图8-6显示了结果。当乘客的数目小于20，我们推断系统运行正常，此时长时间延迟的概率很小。如果有30名乘客，我们估计自上趟火车已经过了15分钟；这比正常延迟时间长，因此我们推断出了某些问题，并预期会有更长的延迟。

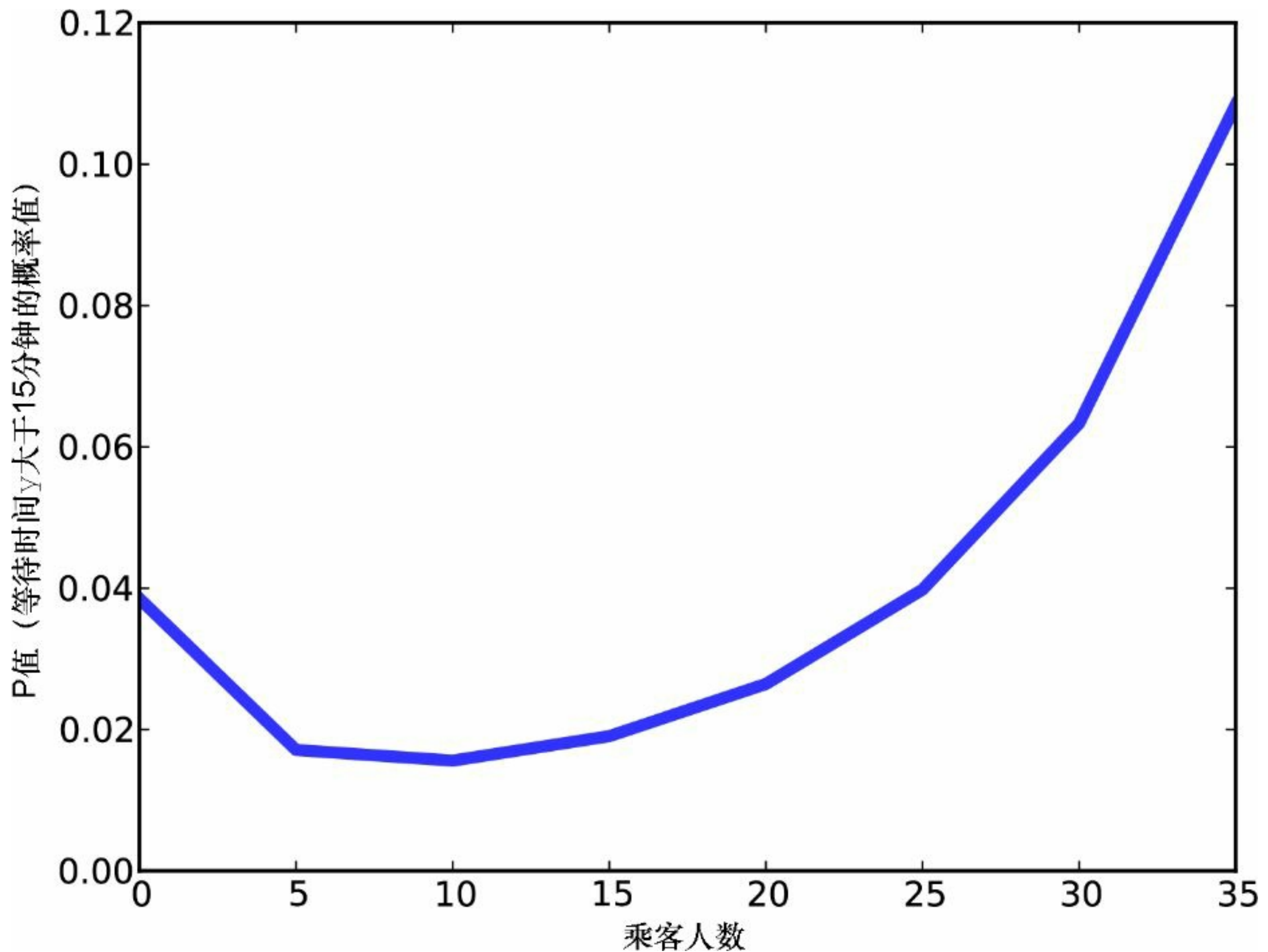


图8-6　以站台上乘客人数为变量的等待时间超过15分钟的概率函数

如果能接受有10%的概率会错过南站列车，又当有不到30名乘客的时候，我们应留下来继续等待。但如果发现乘客更多的话，应选择乘坐出租车。

或者，进一步分析，我们可以量化错过南站列车的成本和乘坐出租车的费用，然后选择最小化预期成本的阈值。

8.8　讨论

分析到目前为止一直基于一个假设，即乘客的到达率每天是相同的，对于高峰时段的通勤列车，这可能不是一个坏假设，但也有一些明显的例外。例如，如果附近有一个特殊的事件，大量的乘客可能同时到达。在这种情况下，`lam` 的估计就会太低，所以`x` 和`y` 的估计会太高。

如果特殊事件和重大延误一样常见，将它们包括进模型就很重要。我们可以通过扩展`lam`的分布以包括进偶尔出现的较大值来实现这一点。

我们是从假设已知`z`的分布开始的。另一个办法是乘客可以估算`z`，但这也不容易。作为乘客，你只能观察到自己的等待时间`y`。除非你略过遇到的第一辆列车，等第二辆到站，否则你就不能直接观测到列车的间隔`z`。

不过，我们可以做出`zb`的一些推论。如果我们注意自己抵达车站时的乘客人数，我们可以估算自上一趟车后所过去的时间`x`，然后观察`y`。如果我们把`x`的后验分布与观测到的`y`相加，就得到了表示`zb`观测值的后验信念的分布。

我们可以利用这个分布来修正我们对`zb`的分布信度。最终可以通过反向计算`BiasPmf`从`zb`得到`z`的分布。

我留下一个分析练习给读者。建议：阅读第15章。你可以在*http://thinkbayes.com/redline.py*找到解法的梗概。欲了解更多信息，请参见前言的“代码指南”。

8.9 练习

练习8-1。

这一练习来自麦凯《信息论、推理和学习算法》一书：

> 有不稳定的粒子从一个源上射出，并于距离源x的位置上衰减，x是含有参数λ的指数概率分布。衰变只有在$x = 1$厘米到$x = 20$厘米的窗口内才能被观察到。如果在距离1.5厘米、2厘米、3厘米、4厘米、5厘米、12厘米处观测到N个衰变，λ的后验分布是什么？

你可以从*http://thinkbayes.com/decay.py*下载这个练习的解法。

第9章　二维问题

9.1　彩弹

彩弹射击运动中，参赛队伍用彩弹互相射击，用涂料填充的彩弹命中时会破碎。这一运动通常在一个安置了障碍和其他可作为掩护物体的区域中进行。

假设你在一个宽30英尺长50英尺的室内场地玩彩弹，靠近一面30英尺的墙壁站着，怀疑你的对手之一到了墙角附近。沿着墙壁，你看到几个有相同颜色的彩弹痕迹，可以认为你的对手刚刚开火了。

彩弹痕迹是在沿着左下角墙角15英尺、16英尺、18英尺和21英尺处。根据这些数据，你认为你的对手躲藏的位置在哪里？

图9-1显示了场地的平面图。以房间的左下角为原点。我以 α 和 β 作为射击方的未知位置坐标，或称为`alpha` 和`beta` 。彩弹痕迹位置被标记为`x` 。对手射击的角度为 ϑ 或`theta` 。

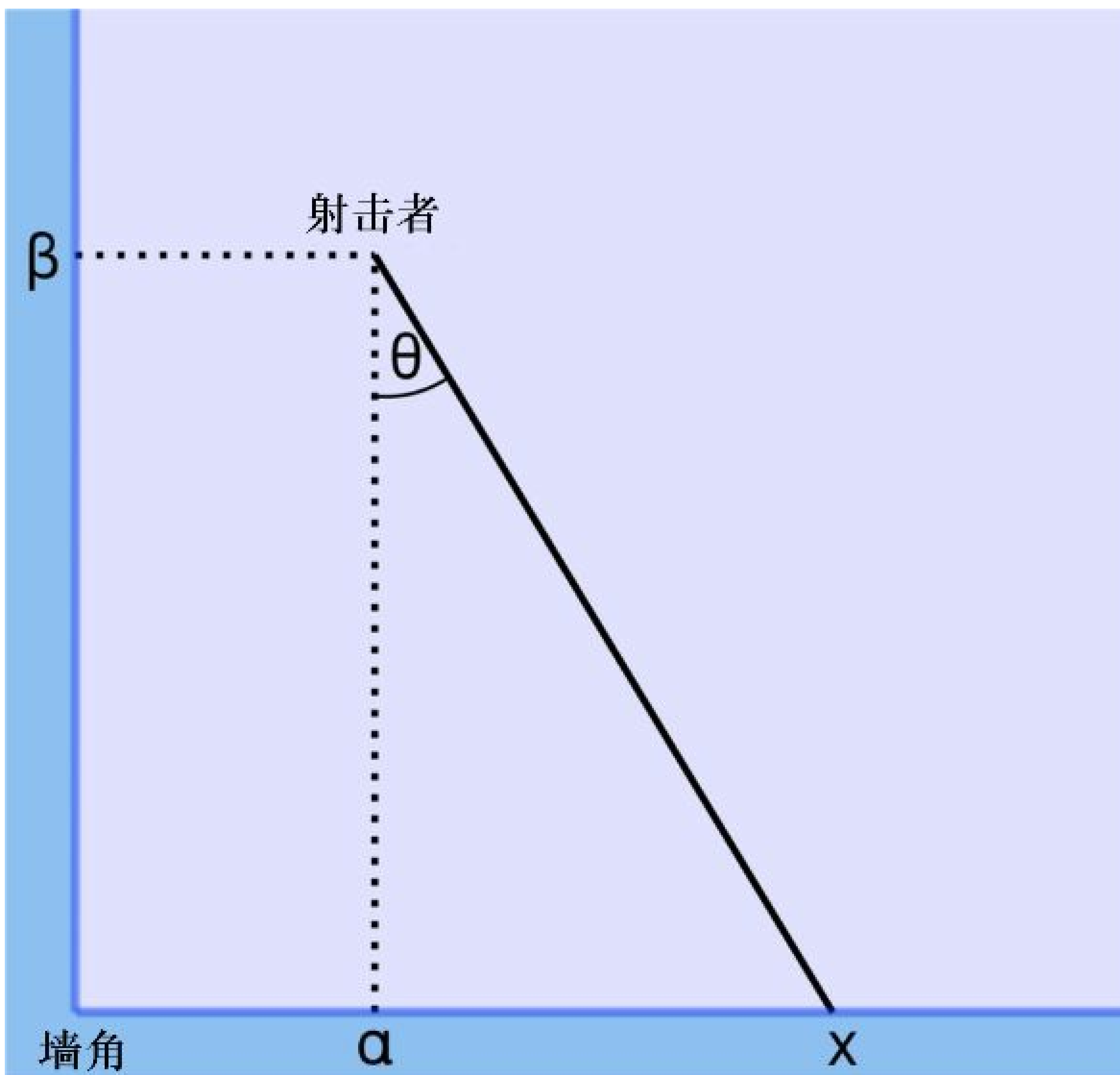


图9-1　彩弹问题布局图

该彩弹问题是贝叶斯分析案例中常见的灯塔问题的一个修改版本，我的标注依据了D.S.Sivia在一《数据分析：贝叶斯教程（第二版）》（牛津出版社，2006年）书中对于该问题的介绍。

你可以从 *http://thinkbayes.com/paintball.py* 下载本章代码。更多信息请参见前言的“代码指南”。

9.2　Suite对象

首先，我们需要一个Suite对象来表示一组有关比赛对手位置的假

设。每个假设是一对坐标：(alpha，beta)。

下面是彩弹Suite对象的定义：

```
class Paintball(thinkbayes.Suite, thinkbayes.Joint):

    def __init__(self, alphas, betas, locations):
        self.locations = locations
        pairs = [(alpha, beta)
                 for alpha in alphas
                 for beta in betas]
        thinkbayes.Suite.__init__(self, pairs)
```

Paintball 继承自我们见过的Suite 对象，至于Joint 我会马上开始介绍。

alphas 是alpha 所有可能值的列表，betas 是beta 值的列表。pairs 是所有的（alpha，bata）对的列表。locations 是沿墙的可能位置列表，它被存储在Likelihood 中以便后面使用。

房间为30英尺宽，50英尺长，下面是一个创建该Suite对象的代码：

```
alphas = range(0, 31)
betas = range(1, 51)
locations = range(0, 31)

suite = Paintball(alphas, betas, locations)
```

这个先验分布假设房间里的所有位置都等可能。已知房间的地图后，我们可选择一个更详细的先验分布，但我们先从简单情况的开始。

9.3　三角学

现在我们需要一个似然函数，这意味着我们必须弄清楚在已知对手的位置后，他击中任意一个沿着墙壁的位置点的似然度。

作为一个简化模型，假设对手像一个旋转着的炮塔，向任意方向射击的可能性相同。在这种情况下，他最有可能打中墙壁的alpha 位置，

击中远离alpha 的位置可能性较小。

利用一些三角学知识，我们可以计算击中任意沿着墙壁点上概率。试想一下，射手以角度θ 射击，彩弹会击中墙上的位置x ，其中

$$x - \alpha = \beta \tan \theta$$

解这个方程得到θ 值

$$\theta = \tan^{-1} \left(\frac{x - \alpha}{\beta} \right)$$

因此，已知墙上某个位置，我们可以求出θ 。

取第一方程相对于 θ 的导数

$$\frac{dx}{d\theta} = \frac{\beta}{\cos^2\theta}$$

这个导数我称之为"扫射速度"，这是目标随着 θ 增加而沿着墙运动的速度（一个映射）。击中墙上一个给定的点的概率和扫射速度负相关。

如果我们知道射手的坐标和墙壁上的一个位置，我们就可以计算出扫射速度：

```
def StrafingSpeed(alpha, beta, x):
    theta = math.atan2(x - alpha, beta)
    speed = beta / math.cos(theta)**2
    return speed
```

alpha 和beta 是射手的坐标；x 是一个彩弹的位置。结果是x 相对于theta 的导数。

现在，我们可以计算出表示击中任何位置概率的Pmf 了。MakeLocationPmf 接收的参数是射手的坐标alpla 和beta ；彩弹命中的位置Locations （一个Locations 可能值的列表）。

```
def MakeLocationPmf(alpha, beta, locations):
```

```
    pmf = thinkbayes.Pmf()
    for x in locations:
        prob = 1.0 / StrafingSpeed(alpha, beta, x)
        pmf.Set(x, prob)
    pmf.Normalize()
    return pmf
```

MakeLocationPmf计算出了击中墙上每个位置的概率，其反比于扫射速度。其结果是位置的Pmf和它们的概率。

图9-2显示了用alpha=10和一系列的beta值计算的Pmf值。对于测试的所有值，最有可能的位置为x = 10；随着beta的增加，Pmf范围也会扩大。

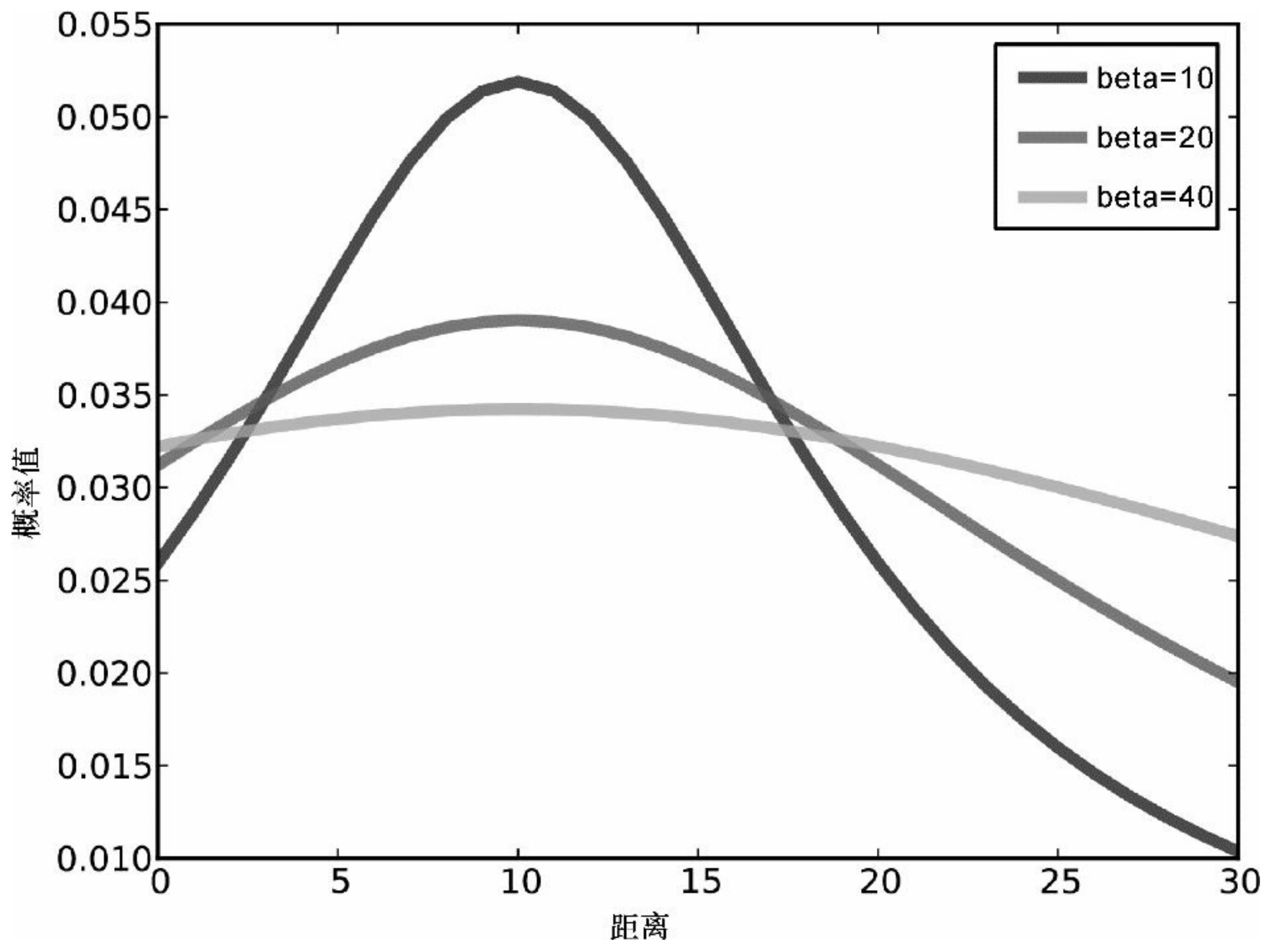


图9-2　用alpha=10和一系列的beta值计算的PMF值

9.4　似然度

现在我们需要一个似然函数。在给定对手的坐标的情况下，我们可以使用MakeLocationPmf 计算任意x 的似然度。

```
    def Likelihood(self, data, hypo):
        alpha, beta = hypo
        x = data
        pmf = MakeLocationPmf(alpha, beta, self.locations)
        like = pmf.Prob(x)
        return like
```

再强调下，alpha 和beta 是射手的假想坐标，x 是彩弹命中痕迹的位置。

pmf 包含了给出射手的坐标时墙上每个位置点的概率。从Pmf我们可以得到观察到的彩弹位置的概率。

大功告成。要更新Suite对象，我们可以使用继承自Suite 的UpdateSet 。

```
suite.UpdateSet ([ 15 ， 16 ， 18 ， 21 ])
```

其结果是一个映射每个（alpha ， beta ）到一个后验概率的分布。

9.5　联合分布

当分布的每个值都是一个元组变量时，被称为联合分布 。它代表了多个变量的分布，这正是“联合”的含义。联合分布包含了变量的分布以及变量间的关系。

给定一个联合分布，我们可以计算每个变量的独立分布，这被称为边缘分布（ marginal distribution）。

thinkbayes.Joint 提供了计算边缘分布的方法：

```
# class Joint:

    def Marginal(self, i):
        pmf = Pmf()
        for vs, prob in self.Items():
            pmf.Incr(vs[i], prob)
        return pmf
```

i 是我们想要的变量的索引，在该示例中i = 0 表示alpha 分布，i = 1 表示beta 分布。

下面是提取边缘分布的代码：

```
    marginal_alpha = suite.Marginal (0)
    marginal_beta = suite.Marginal (1)
```

图9-3显示了结果（转换为CDFs）。对于alpha，中值是18，就在观察到的彩弹数据集的中心附近。对于beta，最可能的值靠近墙壁（<10），而10英尺外的分布几乎是均匀的，这恰好表明了在这些可能位置间的数据非常不起眼（即可能性不高）。

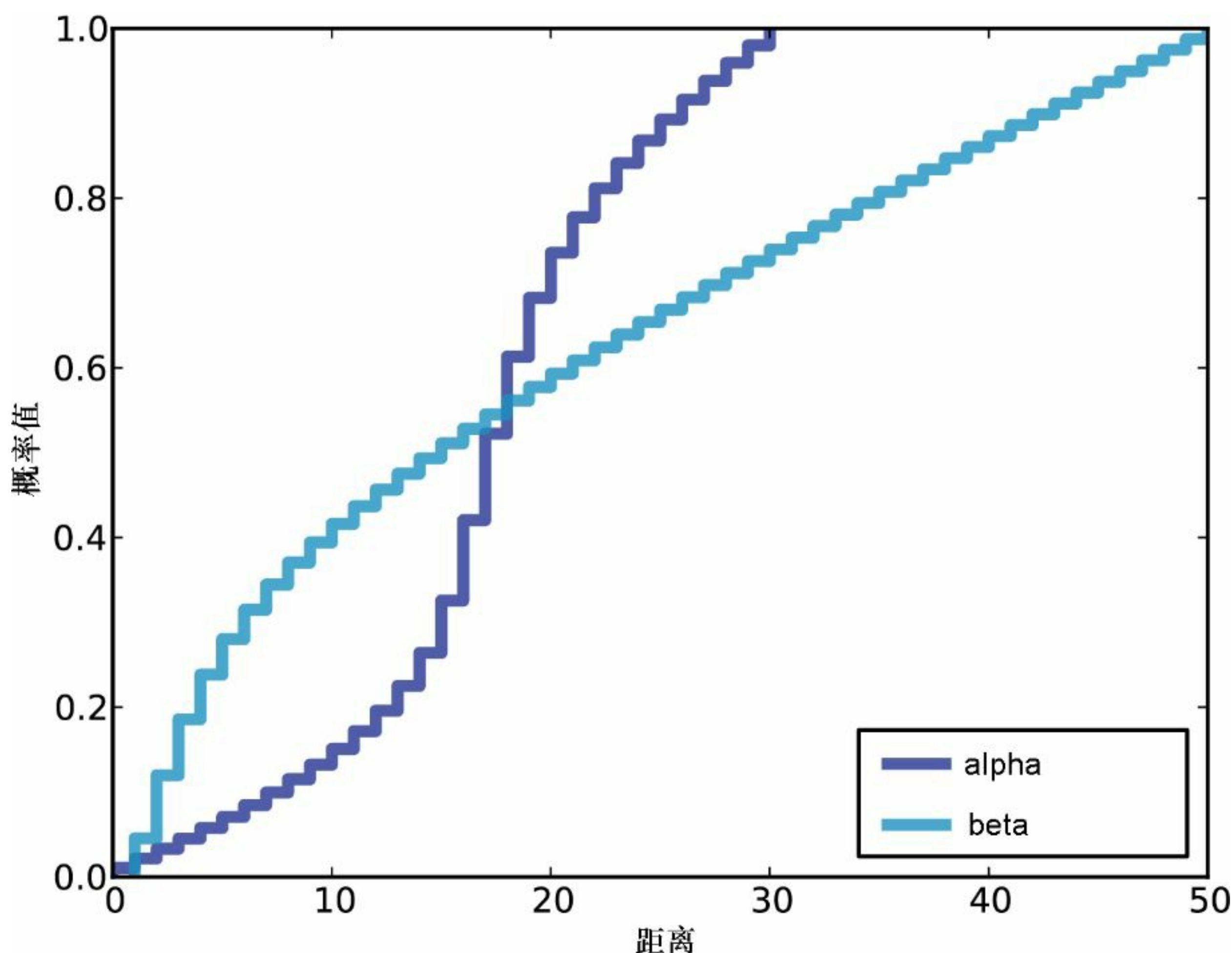


图9-3　已知数据后，alpha 和beta 的后验CDF曲线

给定的后验的边缘，我们可以分别为每个坐标计算置信区间：

```
print 'alpha CI ' , marginal_alpha.CredibleInterval(50)
print 'beta CI ' , marginal_beta.CredibleInterval(50)
```

对于alpha，50%置信区间为(14，21)，beta 是(5，31)。所以数据提供的证据表明射手处在房间内的近侧。但这还不是强证据，因为90%置信区间已经覆盖了房间的大部分区域！

9.6　条件分布

边际分布表示了有关变量各自的信息，但它没有捕捉变量之间的依赖关系。

可视化其依赖关系的一种方法是通过计算条件分布。`thinkbayes.Joint` 提供了一个方法：

```
    def Conditional(self, i, j, val):
        pmf = Pmf()
        for vs, prob in self.Items():
            if vs[j] != val: continue
            pmf.Incr(vs[i], prob)

        pmf.Normalize()
        return pmf
```

同样的，`i` 是我们想要的变量的索引；`j` 是调节变量的索引，`val` 是有条件值。

其结果是第 *i* 个变量在第 *j* 个变量值是`val` 这一条件下的下的分布。

例如，下面的代码计算`alpha` 在一定`beta` 范围下的分布：

```
betas = [10, 20, 40]

for beta in betas:
    cond = suite.Conditional(0, 1, beta)
```

图9-4显示了结果，我们可以将其完整地表述为“条件边缘的后验分布”（posterior conditional marginal distribution）。

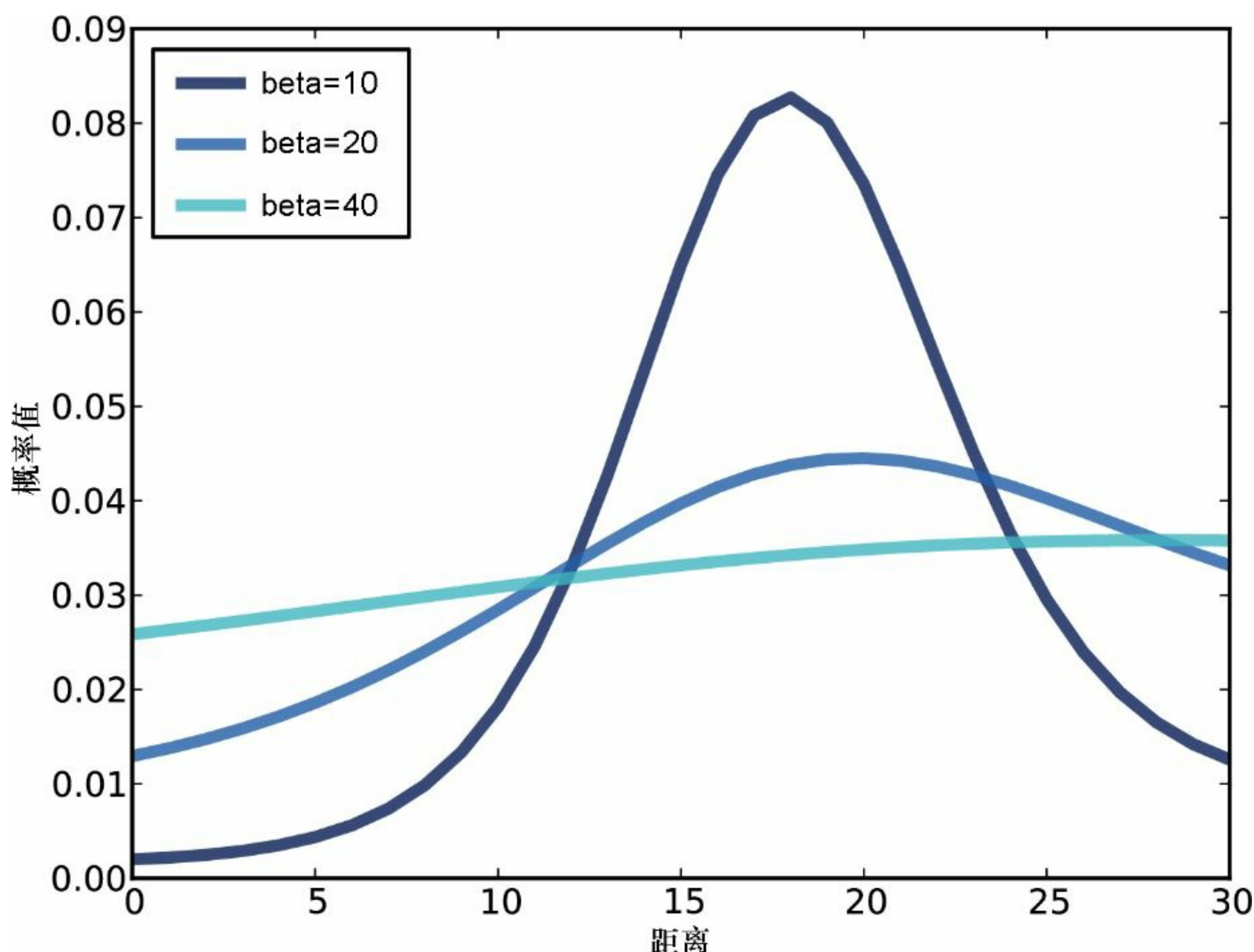


图9-4　以数个beta数据为条件的，alpha的后验概率分布曲线

如果该变量是独立的，条件分布就会相同。但因为它们（条件分布）不同，我们因此可以说变量是相关的。

例如，如果已知`beta = 10`，`alpha`的条件分布范围就相当狭窄。而对于较大的`beta`值，`alpha`的分布范围就较宽。

9.7　置信区间

另一种可视化后验联合分布的方式是计算置信区间。当我们在第23页讨论“置信区间”的时候，我略过了一个微妙信息点：对于给定的分布，有很多具备相同置信度的区间。例如，如果你想有一个50%的可信区间，你可以选择任意一组值的累加起来是50%的概率值。

当该值是一维的，最常见的是选择中心的置信区间，例如，中央

50%置信区间包含了第25和第75百分位数的所有值。

在多个维度情况下，什么是正确的置信区间并非显而易见。最好的选择可能要取决于上下文，但一个常用的选择是最大似然置信区间，其中包含了累加为 50%的最有可能的值（或一些其他百分比数）。

thinkbayes.Joint 提供了计算最大似然置信区间的方法：

```
# class Joint:

    def MaxLikeInterval(self, percentage=90):
        interval = []
        total = 0

        t = [(prob, val) for val, prob in self.Items()]
        t.sort(reverse=True)

        for prob, val in t:
            interval.append(val)
            total += prob
            if total >= percentage/100.0:
                break

            return interval
```

第一步是在Suite中创建一个包含所有值的列表，以概率值递减顺序存储。接着遍历列表，从高到低累加概率直到超过预设百分比 percentage 。这一步的结果是从Suite对象得到的值列表。请注意，这组值不一定是连续的。

要可视化这些区间，我写了一个函数，根据每个区间的出现次数为每个概率值“着色”：

```
def MakeCrediblePlot(suite):
    d = dict((pair, 0) for pair in suite.Values())

    percentages = [75, 50, 25]
    for p in percentages:
        interval = suite.MaxLikeInterval(p)
        for pair in interval:
            d[pair] += 1
```

```
    return d
```

d 是一个在Suite中将每个概率值和其所在区间次数进行映射的字典。循环部分的代码进行这些区间计算并修改d 。

图9-5显示了结果。25%置信区间为靠近墙沿的最暗区域。对于更大的百分比，置信区间更大并且偏向房间的右侧。

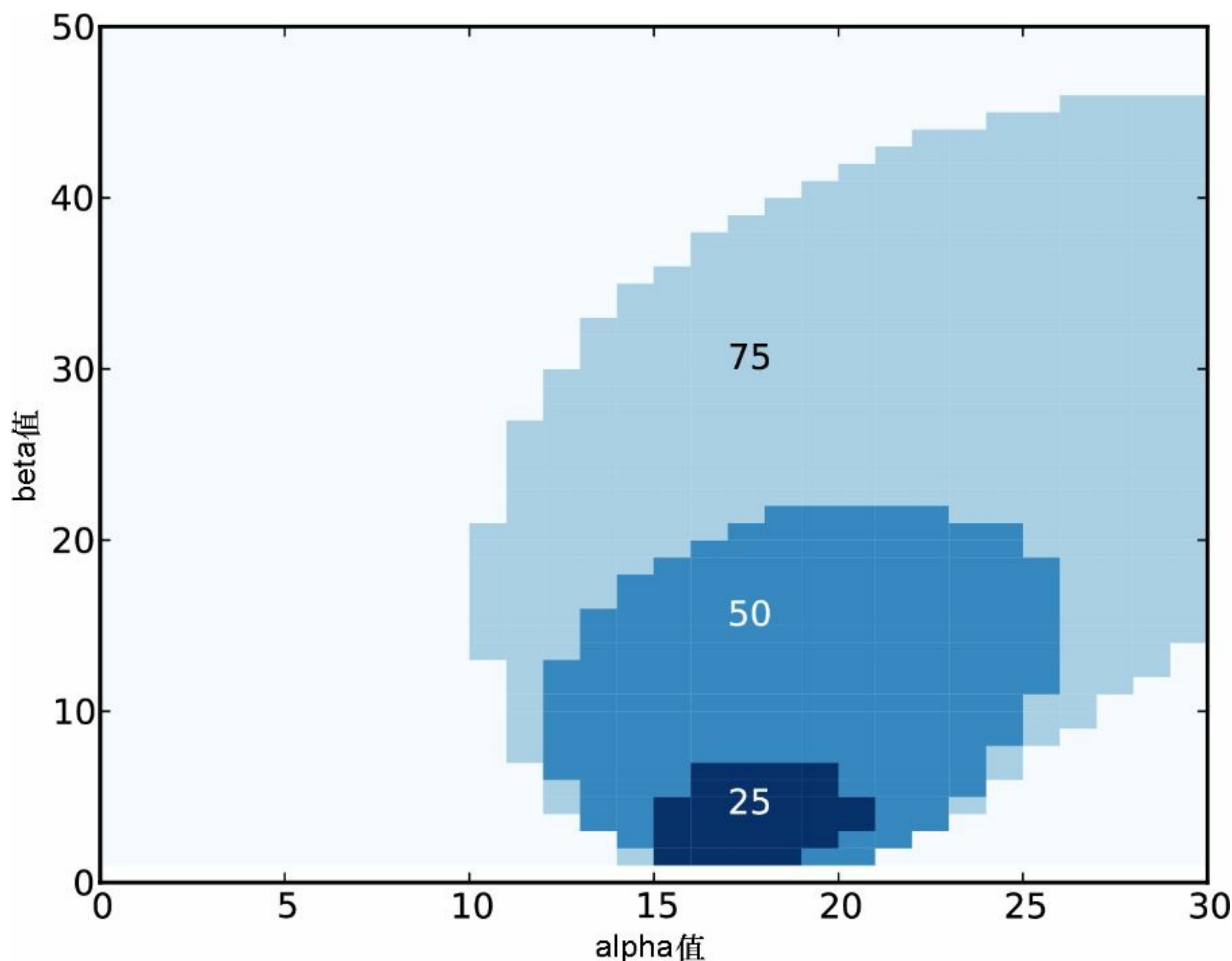


图9-5　对手坐标的置信区间

9.8　讨论

本章的内容说明了前几章里提到的贝叶斯框架可扩展到处理二维参数空间。唯一的区别在于，每个假设由参数元组来表示。

我还介绍了 Joint 对象，这是一个提供了联合分布计算方法的父类，可用于计算Marginal，Conditional，还有MakeLikeInterval 等联合分布。以面向对象的概念来说，Joint 对象是一个混入对象（见 *http://en.wikipedia.org/wiki/Mixin* ）。

本章中出现了很多新的词汇，让我们来回顾下。

联合分 布：

表示在多维空间中所有可能值和它们的概率。本章中的例子是由坐标alpha 和beta 构成的二维空间。联合分布代表了每一个（alpha ，beta ）对的概率。

边缘分布：

联合分布中，某个参数在其他参数未知情况下的分布，例如图9-3 展示的各自独立的alpha 分布和beta 分布。

条件分布：

在一个联合分布中。给定一个或者多个其他参数条件的情况下，某个参数的分布。例如，图9-4显示了alpha 在beta 的几个不同值下的几个分布。

已知联合分布，可以计算出边际和条件分布。有了足够的条件分布信息，可以至少是近似地重建联合分布。但已知边际分布不能重新建立联合分布，因为已经丢失了变量之间相关性的信息。

如果对两个参数都有n 个可能的值，在联合分布中的大多数操作需要的是与n^2 成正比的运算时间。如果还有参数d ，运行时间正比于n^d ，而且随着维数的增加，迅速地变得不实用。

如果你要在合理时间内处理上百万个假设值，这些情况是可行的：2个维度，每个参数有1000个可能值；或3个维度，每个参数100个可能值；或6个维度，每个参数10个值。

如果需要处理更多维度，或每个维度上更多值，可以试试其他的优化。我在第15章提供了一个例子。

你可以从 *http://thinkbayes.com/paintball.py* 下载代码。更多信息，请参见前言的“代码指南”。

9.9　练习

练习9-1。

在我们的简单模型中，对手从任何方向射击都有同等的可能。作为练习，我们来考虑改进这一模型。

本章的分析表明射手最有可能对墙的近端射击。但在现实中，如果对手靠近墙壁，因为他不可能看到自己和墙壁之间的目标，所以也不大可能射击到墙上。

设计一个改进的模型，将这种行为考虑在内。尝试找到一种更真实的模型，但不要太复杂。

第10章　贝叶斯近似计算

10.1　变异性假说

我对古怪科学有些偏爱。最近我访问了Norumbega塔，这是一个埃本·诺顿·霍斯福德奇怪理论的不朽纪念碑，它是双效发酵粉和赝品史的发明者。不过这不是本章要讨论的。

本章是关于变异性假说，即

> “最早在19世纪初叶，约翰·梅克尔认为男性的能力范围大于女性，尤其在智商上。换句话说，他相信天才和弱智人口中，男性占绝大多数。梅克尔断言女性缺少变异性正是较为低级的表现，所以他觉得男性是‘优越生物’。”
>
> 来自 *http://en.wikipedia.org/wiki/Variability_hypothesis* 。

我尤其喜欢最后那部分，因为我怀疑，如果事实能证明女性的变异性实际上更大，梅克尔也会把这作为较低级的标志。无论如何，你总能听到变异性假说的证据站不住脚的说法。

不过，最近我在课上检查来自CDC的危险因素监测系统（BRFSS）的数据，尤其是自我提交的成年美国男性和女性的高度数据时，也遇到了这个问题。该数据集包括了154407名男性和254722名女性。以下是我们的发现：

- 男性平均身高为178厘米；女性平均身高为163厘米，所以从平均值看男人高大，这毫不奇怪。
- 男性身高数据的标准偏差是7.7厘米；女性是7.3厘米，因此，从绝对值上看，男性身高的变异性更大些。
- 但是为了比较组间数据的变异性，采用变异系数（Coefficient of Variation，CV）更有意义，即标准偏差除以平均值。相对于刻度而言，这是变异性的一个量纲无关的衡量值。对于男性CV为0.0433；对于女性CV是0.0444。

这就很靠谱了，所以我们可以得出结论，此数据集提供了有力的证据反对变异性假说。但是我们还可以用贝叶斯方法得到更精确的结论，在回答这个问题的过程中，我还有机会来演示一些处理大数据集的技巧。

我以下面几个步骤进行：

1．我们先从最简单的实现开始，不过它仅适用于小于1000个值的数据集。

2．通过对概率进行对数变换，我们可以扩展到全量数据集，但计算会变慢。

3．最后，我们以近似贝叶斯计算加快计算过程，它也被称为ABC（Approximate Bayesian Computation）。

你可以从 *http://thinkbayes.com/variability.py* 下载本章中的代码。欲了解更多信息，请参见前言的“代码指南”。

10.2 均值和标准差

在第 9 章我们通过联合分布同时估算了两个参数。在本章中，我们用同样的方法来估计高斯分布的参数：均值`mu`，标准差`sigma`。

就这个问题，我定义了一个名为`height`的Suite对象，其表示每一个`mu,sigma`对到其概率的映射：

```
class Height(thinkbayes.Suite, thinkbayes.Joint):

    def __init__(self, mus, sigmas):
        thinkbayes.Suite.__init__(self)

        pairs = [(mu, sigma)
                 for mu in mus
                 for sigma in sigmas]

        thinkbayes.Suite.__init__(self, pairs)
```

mus 是mu 值的一个序列；sigmas 的是sigma 值的一个序列。所有mu ，sigma 对的先验分布是一个均匀分布。

似然函数容易得出，考虑到mu 和sigma 的假定值，我们计算某一个特定值x的似然度。EvalGaussianPdf 的作用，就是这个，因此我们要做的就是使用它：

```
# class Height

    def Likelihood(self, data, hypo):
        x = data
        mu, sigma = hypo
        like = thinkbayes.EvalGaussianPdf(x, mu, sigma)
        return like
```

如果曾经从数学的角度学习过统计学，你知道当进行PDF估计时得到的是概率密度。为了得到一个概率，你必须在一定范围上对概率密度积分。

但就我们的目的来说，我们其实不需要概率，只是需要一些正比于所求概率的量，概率密度就挺合适。这个问题最难的部分是选择mus 和sigmas 的合适范围。如果范围太小，会忽略了一些值得注意的概率值，导致得到错误结果。如果范围太大，尽管可以得到正确的答案，但是白白浪费了计算能力。

所以在这里有机会利用经典的估计方法，让贝叶斯方法更高效。具体说，我们可以用经典的估计量找到mu 和sigma 的最可能位置，并利用这些估计的标准误差来选择一个可能的范围。如果分布的实际参数是μ 和σ ，我们取n 个值的一个样本，μ 的估计量就是样本的均值m 。

σ 的估计量是样本的标准方差，s 。

对μ 的估计的标准误差为$s/\sqrt{n}$ ，而对σ 的估计的标准误差为$s/\sqrt{2(n-1)}$ 。

下面就是所有相关计算的代码：

```
def FindPriorRanges(xs, num_points, num_stderrs=3.0):
```

```
    # compute m and s
    n = len(xs)
    m = numpy.mean(xs)
    s = numpy.std(xs)

    # compute ranges for m and s
    stderr_m = s / math.sqrt(n)
    mus = MakeRange(m, stderr_m)

    stderr_s = s / math.sqrt(2 * (n-1))
    sigmas = MakeRange(s, stderr_s)

    return mus, sigmas
```

xs 是数据集。num_points 是取值范围内所需的值的个数。num_stderrs 是估计量两侧该范围的宽度，以标准误差计。

返回值是mu 和sigma 数值对构成的一个序列。

MakeRange 如下：

```
def MakeRange(estimate, stderr):
    spread = stderr * num_stderrs
    array = numpy.linspace(estimate-spread,
                           estimate+spread,
                           num_points)
    return array
```

numpy.linspace 创建一个由等距分隔的元素构成的数组，在estimate-spread 和estimate+spread 区间上，包括左右两端的值。

10.3　更新

最后，下面是创建和更新Suite对象的代码：

```
mus, sigmas = FindPriorRanges(xs, num_points)
suite = Height(mus, sigmas)
suite.UpdateSet(xs)
print suite.MaximumLikelihood()
```

根据数据来选择先验分布的范围，接着又利用这些数据再次做了更新，这个过程多少有些作伪。通常来说，两次利用了相同的数据，事实上的确是作伪。

但在这个例子里是可以的。的确，我们用数据来选择先验的范围，但这只是为了避免计算数量很大、值却微乎其微的概率。给定num_stderrs = 4，范围就已经足以覆盖所有值得考虑的似然度了，随后再扩大范围对结果也毫无影响。

实际上，mu 和sigma 的所有值上，先验都是均匀分布的，但为了计算效率我们忽略了所有不重要的值。

10.4　CV的后验分布

一旦有了mu 和sigma 的后验联合分布，我们就可以计算出男女的CV分布，自然地，其中一个的概率会超过另外一个的概率。

为了计算CV的分布，我们枚举mu 和sigma 的数值对：

```
def CoefVariation(suite):
    pmf = thinkbayes.Pmf()
    for (mu, sigma), p in suite.Items():
        pmf.Incr(sigma/mu, p)
    return pmf
```

然后我们用thinkbayes.PmfProbGreater 来计算男性有更多变异性的概率。

分析本身很简单，但还有两个额外的我们必须处理的问题。

1．随着数据集大小的增加，我们碰到了一系列由于浮点运算限制带来的计算问题。

2．该数据集包含了一些几乎肯定不对的极端值。我们需要让估计过程在遇到这些异常值时也是健壮的。

以下各节将解释这些问题及其解决方案。

10.5　数据下溢

如果从BRFSS数据集选择前100个值进行我所说的分析，程序将会运行正常，而且得到看起来合理的后验分布。

如果我们选择前1000个的值，然后再次运行程序，我们将从Pmf.Normalize 得到一个错误：

```
ValueError: total probability is zero.
```

这个问题在于我们使用概率密度来计算似然度，而连续分布密度往往很小。如果你把1000个值相乘，其结果将是非常小的。在这个例子里，其值甚至小到不能由一个浮点数来表示，所以被向下舍入到零，这被称为下溢 。而如果分布的所有概率是0，也就不再成其为一个分布。

一种可能的解决方案是每次更新后重新归一化Pmf，或者每批计算100个值，这样就正常了，但很慢。

一个更好的选择是以对数变换来计算似然度。这样一来就不是将较小值相乘，而是将对数值相加了。Pmf 提供了Log 、LogUpdateSet 和Exp 方法，使这个过程变得简单。

Log 计算Pmf 中概率的对数：

```
# class Pmf

    def Log(self):
        m = self.MaxLike()
        for x, p in self.d.iteritems():
            if p:
                self.Set(x, math.log(p/m))
            else:
                self.Remove(x)
```

在应用Log 进行对数变换前，使用MaxLike 找到m ——Pmf中最高

的概率值。把所有概率除以m，所以最高概率被归一化为1，这将得到一个为0的对数。其他日志概率均为负值。如果Pmf中有任何值的概率为0，去掉它们。

然而Pmf是基于对数变换的，我们不能使用Update、UpdateSet或Normalize。否则其结果将没有意义；如果你这么用，Pmf会引发异常。相反，我们必须使用LogUpdate和LogUpdateSet。

下面是LogUpdateSet的实现：

```
# class Suite

    def LogUpdateSet(self, dataset):
        for data in dataset:
            self.LogUpdate(data)
```

LogUpdateSet遍历数据并调用LogUpdate：

```
# class Suite

    def LogUpdate(self, data):
        for hypo in self.Values():
            like = self.LogLikelihood(data, hypo)
            self.Incr(hypo, like)
```

LogUpdate类似Update，不同的是它调用Loglikelihood而不是Likelihood，Incr而不是Mult。

采用对数似然避免了下溢的问题，但同时Pmf只是进行对数变换，没有更多作用了。我们还必须使用Exp来转置这一变换过程：

```
# class Pmf

    def Exp(self):
        m = self.MaxLike()
        for x, p in self.d.iteritems():
            self.Set(x, math.exp(p-m))
```

如果对数似然度是一个大的负数，得到的似然值有可能下溢。所以Exp 寻找对数似然度的最大值m ，再用m 转换回所有的似然值（非对数）。得到的分布具有1的最大似然度。这个过程以最小的精度损失转置了对数变换。

10.6 对数似然

现在我们需要的是Loglikelihood 。

```
# class Height

    def LogLikelihood(self, data, hypo):
        x = data
        mu, sigma = hypo
        loglike = scipy.stats.norm.logpdf(x, mu, sigma)
        return loglike
```

norm.logpdf 计算高斯PDF的对数似然值。

下面是整个更新过程：

```
    suite.Log ()
    suite.LogUpdateSet (xs)
    suite.Exp ()
    suite.Normalize ()
```

回顾一下，Log 将Suite对象进行对数转换。LogUpdateSet 调用LogUpdate ，再调用Loglikelihood 。LogUpdate 使用Pmf.Incr ，因为将对数似然值相加等同于似然度相乘。

更新后，对数似然都是较大的负数，所以在进行变换转置前，Exp 对它们进行转换，这就是我们避免下溢的过程。

一旦该 suit 对象被转化回来，概率就变回“线性”的，这意味着其“非对数”了，所以我们可以再次使用Normalize 。

使用这种算法，可以在处理整个数据集时避免产生下溢，但它仍然

很慢。我的电脑处理起来可能需要一小时。我们还可以做得更好。

10.7　一个小的优化

本节通过数学和计算的优化百倍地加快了速度。但下一节介绍了一种还要更快的算法。所以，如果你想直接得到好方法，那么可以跳过这一节。

`Suite.LogUpdateSet` 在每个数据点上调用`LogUpdate` 一次。我们可以通过每一次计算整个数据集的对数似然值来加快步伐。

我们将开始于高斯的PDF：

$$\frac{1}{\sigma\sqrt{2\pi}}\exp\left[-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2\right]$$

并计算对数log（去掉了常数项）：

$$-\log\sigma-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2$$

对于给定的值序列x_i，总对数似然是

$$\sum_i -\log\sigma-\frac{1}{2}\left(\frac{x_i-\mu}{\sigma}\right)^2$$

去掉不包括i 的项，得到

$$-n\log\sigma-\frac{1}{2\sigma^2}\sum_i(x_i-\mu)^2$$

这一过程转换为Python代码如下：

```
# class Height

    def LogUpdateSetFast(self, data):
        xs = tuple(data)
        n = len(xs)
```

```
        for hypo in self.Values():
            mu, sigma = hypo
            total = Summation(xs, mu)
            loglike = -n * math.log(sigma) - total / 2 / sigma**2
            self.Incr(hypo, loglike)
```

本身这只是一个小的优化，但它带来了一个更大的优化可能性。请注意，求和只取决于`mu`，而不是`sigma`，所以对`mu`的每一个值我们只需要计算它一次。

为了避免重新计算，我分解出一个计算总和的函数，并**memoize**它，使其在字典中存储之前的计算结果（见*http://en.wikipedia.org/wiki/Memoization*）：

```
def Summation(xs, mu, cache={}):
    try:
        return cache[xs, mu]
    except KeyError:
        ds = [(x-mu)**2 for x in xs]
        total = sum(ds)
        cache[xs, mu] = total
        return total
```

`cache`存储先前计算的总和。如果可能`try`语句从cache中返回一个结果，否则计算总和，再缓存并且返回计算结果。

唯一美中不足的是，我们不能用列表作为缓存中的一个key，因为它不是一个哈希类型。这就是为什么`LogUpdateSetFast`将数据集转换为一个元组。

这种优化以大约100的因数加快了计算，在我不算快的计算机上处理整个数据集（154 407名男性和254 722名女性）用了不到一分钟。

10.8　ABC（近似贝叶斯计算）

但是，也许你耗不起这样的时间。这时候，近似贝叶斯计算（ABC）就是合适的方法了。ABC背后的动因是，任何特定数据集的似

然度有以下特点。

1．它非常小，特别是对于大型数据集来说，这就是为什么我们必须使用对数转换形式的原因。

2．计算开销大，所以我们不得不做这么多的优化。

3．它并非我们实际所要求的。

我们并不真正关心看到某一已知数据集的具体似然度。尤其对于连续变量，我们关心的是观测到一个类似于已知数据的数据集的似然度。例如，在欧元问题上，我们不关心硬币翻转的顺序，只关心正面和反面的总数。而在火车头问题上，我们不关心看到哪一个具体的列车，而是列车的数量和列车序号的最大值。

同样，BRFSS的样本中，我们并不真想知道看到某一个数据值集合的概率（特别是因为有成千上万的人），而是更类似于提出这样的问题“如果从全体人口中取出一个参数为μ 和σ 的10万人口样本，那么取到一个符合已知观测均值和方差的样本的机会是多少？”

对于一个高斯分布的样本，因为可以分析性地找到样本分布的统计量，我们可以有效地回答这个问题。在计算先验分布的范围时，我们实际已经做到了这一点。

如果从参数为μ 和σ 的高斯分布取n 个值，并且计算样本均值m ，m 的分布是参数为μ 和$\sigma/\sqrt{n}$ 的高斯分布。

同样地，样本的标准差分布s ，也是为参数σ 和$\sigma\sqrt{2(n-1)}$ 的高斯分布。

给定μ 和σ 假设值，我们可以使用这些样本的分布来计算样本统计量m 和s 的似然度。下面是LogUpdateSet 实现这个功能的新版本：

```
def LogUpdateSetABC(self, data):
    xs = data
    n = len(xs)

    # compute sample statistics
    m = numpy.mean(xs)
    s = numpy.std(xs)
```

```
    for hypo in sorted(self.Values()):
        mu, sigma = hypo

        # compute log likelihood of m, given hypo
        stderr_m = sigma / math.sqrt(n)
        loglike = EvalGaussianLogPdf(m, mu, stderr_m)

        #compute log likelihood of s, given hypo
        stderr_s = sigma / math.sqrt(2 * (n-1))
        loglike += EvalGaussianLogPdf(s, sigma, stderr_s)

        self.Incr(hypo, loglike)
```

在我的电脑这个函数处理整个数据集用时大约1秒，得到的结果有5位数的精度与实际结果一样。

10.9 估计的可靠性

我们已经差不多可以看到结果了，但还有一个问题要处理。数据集中有一些几乎肯定是错误的异常数据值。例如，有3个男人身高为61 厘米，也就是说他们是世界上最矮的成年人了。另外，有四个身高229厘米的女性，这个数据仅比世界上最高的女人矮一点。

这些值也不是完全没可能，但还是有点不确定性的，这些因素让处理这些值有些难度。而且我们必须处置得当，因为这些极端值对估计变异性问题有不成比例的影响。

由于ABC基于汇总统计，而不是整个数据集，我们可以选择在有异常值的情况下也稳定的汇总统计量使这一过程更可靠。例如，我们可以不使用样品平均值和标准偏差，而用中位数和四分位数间距（IQR），它在第25和第75个百分位数之间。

更一般地，我们可以计算出一个分布所有区间的百分位间距（IPR）：

```
def MedianIPR(xs, p):
    cdf = thinkbayes.MakeCdfFromList(xs)
    median = cdf.Percentile(50)
```

```
    alpha = (1-p) / 2
    ipr = cdf.Value(1-alpha) - cdf.Value(alpha)
    return median, ipr
```

xs 是一个值序列，p 是所希望的范围；例如，p = 0.5 产生四分位数间距。

MedianIPR 的工作原理是计算xs 的CDF，然后提取中位数和两个百分位之间的差。

通过高斯CDF计算给定标准差的分布的百分数，我们可以从ipr 转换到一个sigma 的估计值。例如，一个众所周知的经验规则是，高斯分布的68%落入均值的一个标准差内，在左右端各留下了其余的16%。如果我们在第16和第84百分位数之间的范围内进行计算，我们预期的结果是2*sigma ，所以我们可以通过计算68%的IPR再除以2来估计sigma 。

更一般地，我们可以选择sigma 的个数。MediaS 提供了这一计算的更通用的版本：

```
def MedianS(xs, num_sigmas):
    half_p = thinkbayes.StandardGaussianCdf(num_sigmas) - 0.5

    median, ipr = MedianIPR(xs, half_p * 2)
    s = ipr / 2 / num_sigmas

    return median, s
```

同样，xs 是值序列；num_sigmas 是结果所决定的标准差数量。其结果是median ，μ 的估计值；还有s ，σ 的估计值。

最后，在LogUpdateSetABC 我们可以用median 和s 代替样品平均值和标准差，效果很好。

这似乎有点儿奇怪，我们正在使用观察的百分位数估计μ 和σ ，但它是贝叶斯方法灵活性的一个示例。实际上，我们一直都在问“给定一个μ 和σ 的假设值，还有一个有可能引入错误的采样过程，那么生成一组给定统计样本的似然度是多少？”

我们可以不受限制地选择任何顺眼的样本统计量，此时：μ 和σ 确定了分布的位置和区间，所以我们需要选择那些表现了这些特征的统计量。例如，如果我们选择了第49和第51百分位数，有关分布区间范围的信息就很少（译注：太窄），所以使得对σ 的估计对数据的约束小。就生成的观测值而言，所有`sigma`的可能值都将具备几乎相同似然度，所以西格玛的后验分布看起来就和前验分布没有区别。

10.10　谁的变异性更大？

我们终于可以回答开头的问题了：男性的变异系数比女性更大吗？

通过使用基于中位数和IPR的ABC方法，假设`num_sigmas = 1`，我计算了`mu`和`sigma`联合分布的后验。图10-1和图10-2显示了结果的等高线图，`mu`在x 轴，`sigma`在y 轴，概率以z 轴表示。

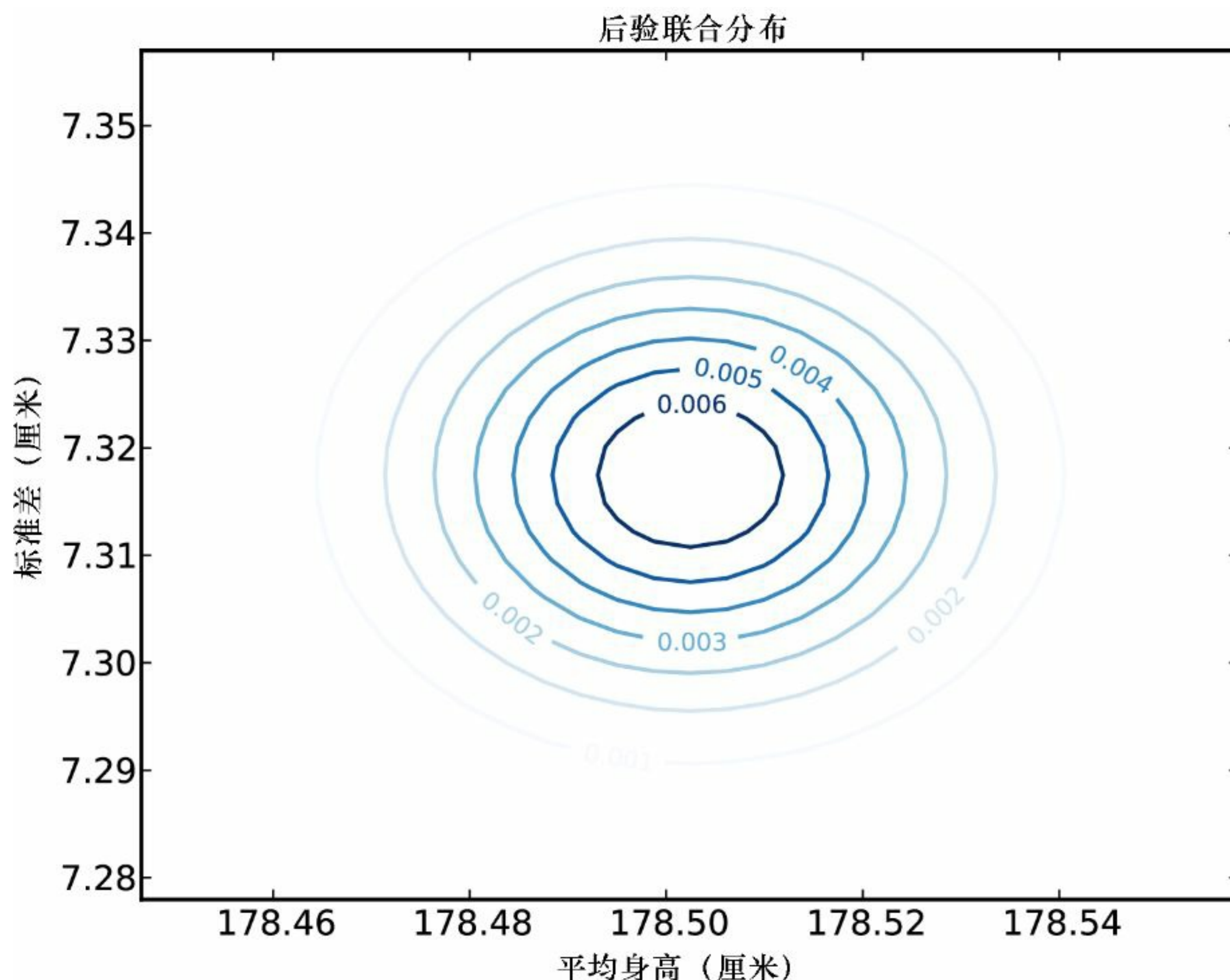


图10-1　美国男性身高平均值和标准差后验联合分布的等高线图

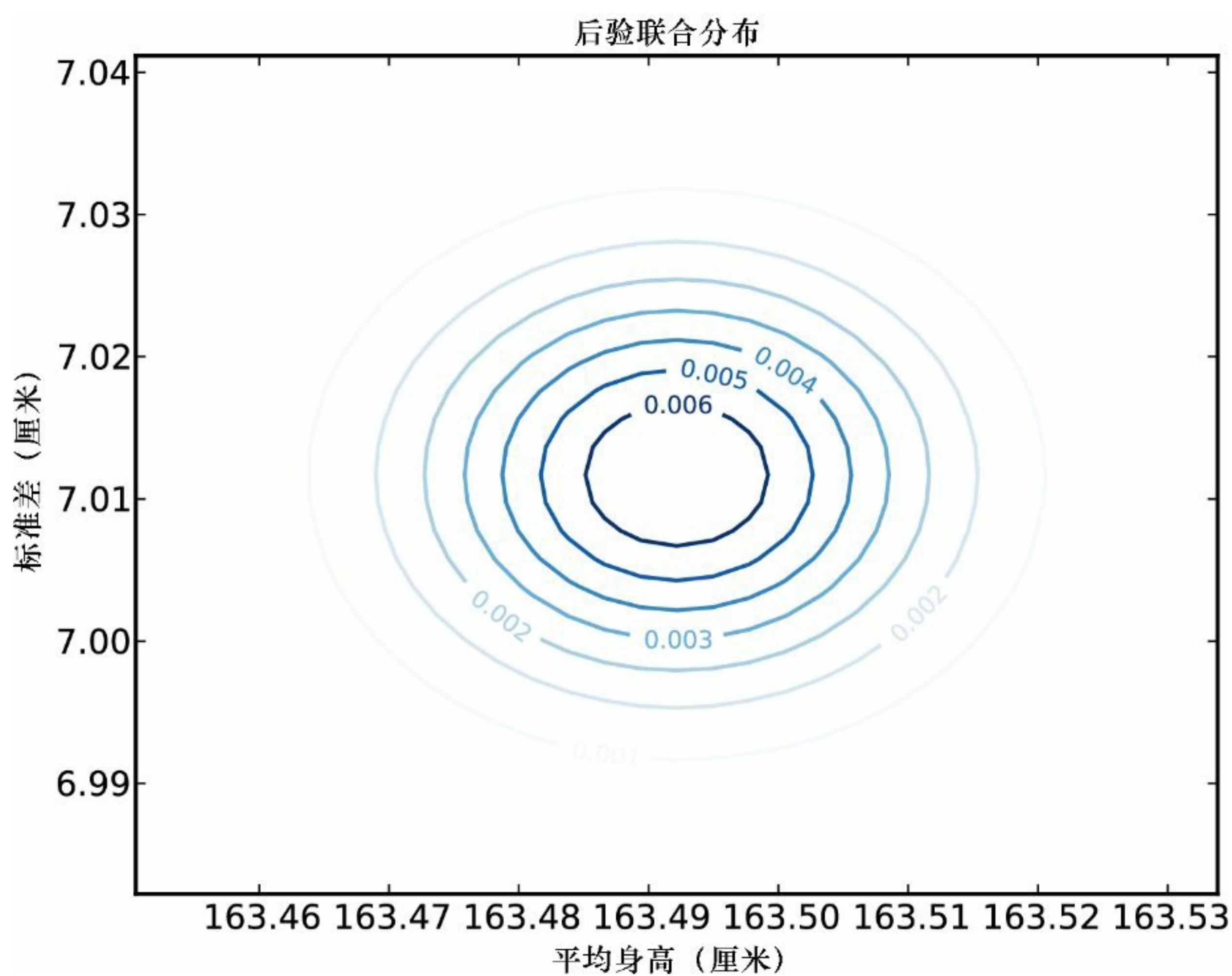


图10-2　美国女性身高平均值和标准差后验联合分布的等高线图

对于每一个联合分布，我计算了CV的后验分布。图10-3显示了男性和女性的这些分布结果。男性平均值为0.0410；女性的平均值为0.0429。由于两者间没有重叠，我们可以比较确定地得出女性在身高方面比男性变异性更大的结论。

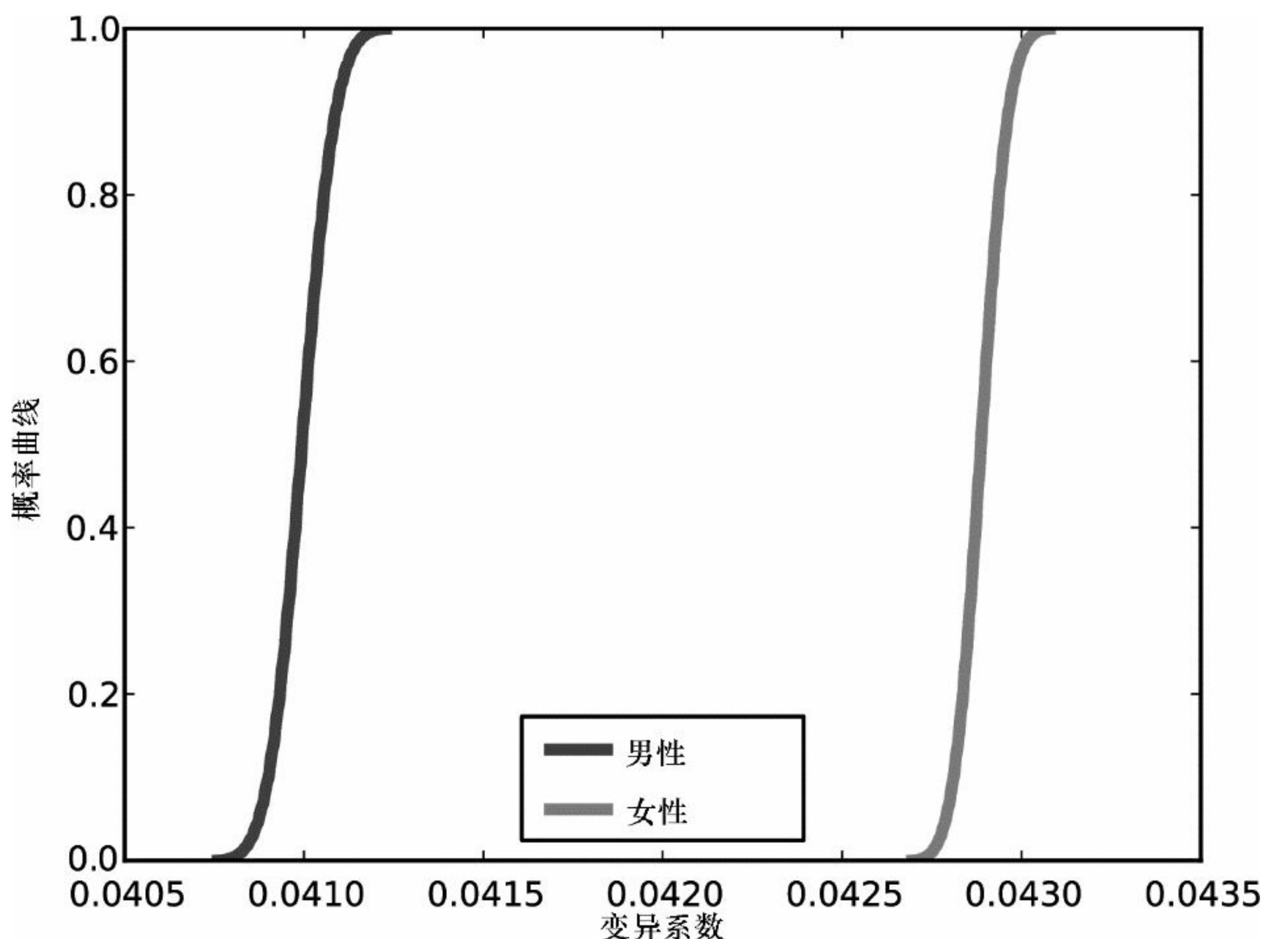


图10-3　男性和女性变异系数CV的后验分布CDF曲线，基于可靠型估计

那么，这就是变异性假设的最终答案了？可悲的是，没有。事实证明，这样的结果依赖于跨百分数范围的选择。指定`num_sigmas = 1`，我们可以得出结论说女性波动范围更大，但若是指定`num_sigmas = 2`，在同等置信度下，结论是男性范围更大。

这一差别的原因在于，矮小身材的男人更多，偏离平均值也较大。

因此，我们对变异性假设的评价取决于对"变异性"的解释。指定`num_sigmas = 1`时，我们关注于接近平均值的人。当加大`num_sigmas`，就给予了极端值更多的权重。

选择究竟要突出问题的哪一面，我们就需要对这一假设更精确的解释。正因为如此，变异性假设可能模糊到难以评价。

然而，这有助于说明我的一些新想法，而且我想你会认可，这是一

个有趣的例子。

10.11　讨论

还有两种对ABC的看法。一种解释是，如名称所指的，和采用实际值的计算相比，近似方法计算起来更快。

但请记住，贝叶斯分析总是基于模型决策的，这意味着不存在“精确”的解决方案。任何惹人关注的物理系统都可能存在许多模型，每个模型产生不同的结果。要对结果进行解释，就必须评估模型。

因此，ABC的另一种解释是，它代表似然度的另外一类模型。当计算p(D |H)，我们提出的问题是“在一个给定假设下，数据的似然度是多少？”

对于大型数据集，数据的似然度非常小，这意味上面的问题可能就不合适。我们真正想知道的结果是新的数据类似于已知数据的可能性，而这里，“类似”的定义又是一个建模决策。

ABC背后的基本思想是，如果两个数据集产生了相同的描述性统计量，那它们就是类似的。但在某些情况下，如在本章的例子中，到底选择哪种汇总统计量并不明显。

你可以从 *http://thinkbayes.com/variability.py* 下载本章中的代码。欲了解更多信息，请参见第前言的“代码指南”。

10.12　练习

练习10-1。

“效应量（effect size）”是一个旨在衡量两组之间的差异的统计量（见 *http://en.wikipedia.org/wiki/Effect_size* ）。

例如，我们可以使用从BRFSS得到的数据去估算男性女性之间的高度差异。由μ 和σ 的后验分布的采样值，就能生成这一差异的后验分布。

但使用效应量的无量纲度量方法可能更好，而不以厘米为单位进行差异衡量。一种选择是通过将其（数据）除以标准差（类似于我们用变异系数一样）。

如果组1的参数为（μ_1，σ_1），组2的参数为（μ_2，σ_2），

无量纲的效应量就是

$$\frac{\mu_1 - \mu_2}{(\sigma_1 + \sigma_2)/2}$$

编写一个函数，它接收两组数据的`mu`和`sigma`的联合分布，并返回效应量的后验分布。

提示：如果枚举两个分布所有数值对的时间太长，应考虑随机采样。

第11章　假设检验

11.1　回到欧元问题

第 27 页的“欧元问题”中，我介绍了来自麦凯《信息、理论、推理和学习算法》中的一个问题：

> 2000年1月4日星期五，《卫报》上刊载了一个统计相关的声明：
>
> 当以边缘转动比利时一欧元硬币250次时，得到的结果是正面140次反面110次。“这看起来很可疑”，伦敦经济学院的统计讲师巴里·布莱特说，“如果硬币是均匀的，得到这个结果的可能性低于7%”。
>
> 那么，这一结果是否为“硬币偏心而非均匀”提供了证据呢？

我们估计了硬币正面朝上的概率，但我们并没有真正回答麦凯的问题：数据是否佐证了硬币是偏心的？

在第 4 章中我提出，如果数据在某一假设下比另外假设下的可能性要高，那么数据就是支持该假设的，这等同于贝叶斯因子大于1的情况。

在欧元问题的例子中，我们考虑两个假设：使用F表示硬币是偏心的假设，B表示硬币是均匀的假设。

如果硬币是均匀的，容易计算数据的似然度，$p(D|F)$。实际上，我们已经完成了一个函数实现这个计算。

```
def Likelihood(self, data, hypo):
    x = hypo / 100.0
    head, tails = data
    like = x**heads * (1-x)**tails
    return like
```

我们可以创建一个调用`Likelihood`的`Euro` suite对象：

```
suite = Euro()
likelihood = suite.Likelihood(data, 50)
```

p(D |F)是5.5×10^{-76}，这一结果除了说明任何特定数据集的概率相当小，没有什么用处。需要求得两个似然度再求它们的比率，所以我们还要计算p(D |B)。

要如何计算B 的似然度并不明确，因为“偏心”的含义并不那么明确。

一种可能性是在定义假设前预先检查数据。那么在这个例子里，“偏心”就是指正面向上的概率为140/250。

```
actual_percent = 100.0 *140/250
likelihood = suite.Likelihood(data, actual_percent)
```

假设B 的这个版本我称之为`B_cheat ; b_cheat`的可能性是34×10^{-76}，似然比是6.1。因此，我们可以说，数据是支持这个版本的B 假设的。

但使用该数据来制定的假设显然是有作伪嫌疑的。根据这一定义，任何数据集都将支持假设B，除非所观察到的正面向上百分比恰好是50%。

11.2　来一个公平的对比

为了实现一个公正的对比，我们必须在无视数据的情况下先定义B 。那么我们来尝试一个不同的定义。如果检查比利时欧元硬币，你可能会注意到“正面”比“反面”更加突出。可以猜想，形状对x 有一定的影响，但不能确定是否就是这一因素让正面多或者少一点。所以，你可能会想：“我认为硬币偏心，所以x 是0.6或0.4，但不知道究竟是多还是少。”

我们可以好好考虑下这个由两个子假设构成的被称为`B_two`的假设

版本。我们可以计算出每个子假设的似然度，然后计算平均似然度。

```
like40 = suite.Likelihood (data，40)
like60 = suite.Likelihood (data，60)
likelihood= 0.5 * like40 + 0.5 * like60
```

对于`b_two` 似然度比（或贝叶斯因子）为1.3，这意味着数据提供微弱的证据支持`b_two` 。

更一般地，设想你怀疑硬币就是偏心不均匀的，可是对于 x 的值没有线索。在这种情况下，你可以建立一个称为`b_uniform` 的Suite对象，代表从0到100的子假设。

```
b_uniform =Euro(xrange (0 ， 101))
b_uniform.Remove (50)
b_uniform.Normalize ()
```

我以值0到100初始化`b_uniform` 。删除了x 等于50%的子假设，因为当x 为50%时，硬币就是均匀的，不管你删不删除除这一值，对结果几乎都没有影响。

要计算`b_uniform` 的可能性，我们计算每个子假设的似然度，并累加其加权平均值。

```
def SuiteLikelihood(Suite, data):
    total = 0
    for hypo, prob in Suite.Items():
        like = suite.Likelihood(data, hypo)
        total += prob * like
    return total
```

`b_uniform` 的似然比是0.47，这意味着相比于F ，数据对`b_uniform` 只算是微弱的证据。

如果你考虑下`SuiteLikelihood` 的计算，你可能会注意到这类似于一个更新过程。来回忆一下，下面是`Update` 的功能：

```
    def Update(self, data):
        for hypo in self.Values():
            like = self.Likelihood(data, hypo)
            self.Mult(hypo, like)
        return self.Normalize()
```

Normalize 如下：

```
def Normalize(self):
    total = self.Total()

    factor = 1.0 / total
    for x in self.d:
        self.d[x] *= factor

    return total
```

Normalize 的返回值是Suite对象中以先验的概率加权的总概率值，即子假设的平均似然度。Update 继续处理这个值，所以不用对SuiteLikelihood 进行处理，我们就可以像下面这样计算b_uniform的似然度：

```
likelihood= b_uniform.Update(data)
```

11.3 三角前验

在第4章中，我们还讨论了形如三角的前验分布，其在50%附近的值有更高的概率。如果将子假设的先验分布设定为三角前验，可以这样计算它的似然度：

```
b_triangle = TrianglePrior()
likelihood= b_triangle.Update(data)
```

和假设F 相比，b_triangle 的似然比为0.84。所以我们可以说该数据对于假设B 只算是微弱的证据。

下表显示了已考虑的可能性的先验概率，以及相对于F的似然比（或贝叶斯因子）。

假设	似然度（$\times 10^{-76}$）	贝叶斯因子
F	5.5	—
`B_cheat`	34	6.1
`B_two`	7.4	1.3
`B_uniform`	2.6	0.47
`B_triangle`	4.6	0.84

根据我们选择的定义，数据会对“硬币是偏心的”提供支持或者反对的证据，但在两种情况下，证据都是相对微弱的。

综上所述，我们可以用贝叶斯假设检验来比较F和B的似然度，但必须做一些工作来精确指出假设B的含义。这一点依赖于硬币和硬币旋转行为的背景信息，所以对所谓正确的定义，人们都有理由提出异议。

我就这个例子的介绍延续了大卫·麦凯的讨论，得到了一样的结论。你可以从 *http://thinkbayes.com/euro3.py* 下载本章代码。欲了解更多信息，请参见前言的“代码指南”。

11.4 讨论

对于`B_uniform`，贝叶斯因子为0.47，和F对比，这意味着数据提供的证据反对这一假设。在前面的部分中，我将这一证据描述为“弱”，但没有说明原因。

部分原因有些年头。哈罗德·杰弗里斯，贝叶斯统计的早期支持

者，提出了一个解读贝叶斯因子的尺度：

贝叶斯因子	强度
1～3	不值一提
3～10	可观的
10～30	强
30～100	很强
>100	决定性的

在本例中，支持`B_uniform`的贝叶斯因子是0.47，所以支持F的贝叶斯因子是2.1，也就是杰弗里斯认为的“不值一提”，其他的学者也提出了不同的用词。若要避免这些形容词上的斟酌，我们可以用胜率来代替。

如果你之前的胜率是1:1，接着你看到了贝叶斯因子为2的证据，你胜率的后验就是2:1。在概率方面，数据将你的信念度（degree of belief）从50%改变到66%。而对于大多数现实世界的问题来说，这一类影响与建模误差和其他不确定性来源导致的影响相比要小些。

另一种情况下，如果你得到一个贝叶斯因子是100的证据，你的后验胜率将是100:1或99%以上。无论你是否同意，这样的证据是“决定性的”，是相当强烈的证据。

11.5　练习

练习11-1。

有些人相信第六感（ESP）的存在。例如，有些人猜扑克牌的能力

比随意进行猜测要高。

在这类ESP上，你的前验信度是什么？你认为它可能存在还是不存在？还是你强烈质疑它呢？写下你的先验胜率。

现在，计算能说服你相信ESP至少有50%可能存在的证据强度。多大的贝叶斯因子能让你90%确定ESP的存在？

练习11-2。

设想前一个问题的答案是1000。也就是说，一个贝叶斯因子是1000的支持ESP的证据才可以改变你的想法。现在假设你在一个权威评审科学期刊上读到了一篇论文，当中提出了一个贝叶斯因子是1000的支持ESP的证据，这会改变你的想法吗？

如果没有，你怎么解决这个明显的矛盾？你会发现阅读大卫·休谟的文章《奇迹》会帮助你思考这个问题。文章见 *http://en.wikipedia.org/wiki/Of_Miracles* 。

第12章　证据

12.1　解读SAT成绩

假设你是马萨诸塞州一个工程学院的招生院长，正在考虑两个候选人爱丽丝和鲍勃，他们在许多方面的资历都差不多，只是爱丽丝在SAT数学部分的得分更高。SAT是旨在衡量大学水平数学准备情况的标准测试。

如果爱丽丝得到了780分，鲍勃得到了 740 分（满分800分），你也许想知道这一差异是否就是爱丽丝比鲍勃准备得更好的证据，还有证据的强度是多少。

现实中，这两个分数都非常好，而且两位候选人可能都为大学数学学习做好了准备。所以真正的院长可能会建议我们选择那些最能体现我们希望学生所具备的技能和态度的候选人。不过既然这是一个贝叶斯假设检验的例子，我们还是坚持住不要扩大问题范畴，即：“爱丽丝比鲍勃准备得更好的证据有多强？”

要回答这个问题，需要进行一些建模决策。我将以一个其实不真实的简化版开始，然后再回来改进模型。暂时的，先假定所有的SAT试题有同等难度。事实上，SAT设计师选择的试题是有一定难度区间的，因为这提高了度量答题者统计差异的能力。

但是，如果我们选择一个全部试题有同等难度的模型，就可以为每个参加测试者定义一个特征`p_correct`，即答对任一问题的概率。这种简化可以很容易地计算出给定得分的似然度。

12.2　比例得分SAT

为了理解SAT成绩，我们要了解得分和比例分数的计算过程。每个测试者基于对题和错题的数量会得到一个原始分数。原始得分被转换为200～800间的比例分数。

2009年，SAT数学部分有54题，原始分数为每个测试者答对题的个数减去答错题的个数乘以1/4分（即，答对1题得1分，答错1题减1/4分）。

负责管理SAT的高校理事会，发布了一个从原始分数到比例分数的映射图。我下载了该数据并将其封装在一个插值对象中，它能进行正向（从原始分数到比例分数）和反向（从比例分数到原始分数）查找。

你可以从 *http://thinkbayes.com/sat.py* 下载这个例子的代码，更多信息请参见前言的“代码指南”。

12.3　先验

美国大学理事会还发布了所有测试者比例分数的分布。如果我们把每一个比例分数转换为原始分数，并除以题目数量，那么结果就是`p_correct`的估计值。所以我们可以使用原始分数的分布为`p_correct`的先验分布建模。

下面是读取并处理数据的代码：

```
class Exam(object):

    def __init__(self):
        self.scale = ReadScale()
        scores = ReadRanks()
        score_pmf = thinkbayes.MakePmfFromDict(dict(scores))
        self.raw = self.ReverseScale(score_pmf)
        self.prior = DivideValues(raw, 54)
```

`Exam`封装了我们已知的有关考试的信息。`ReadScale`和`ReadRanks`读取文件并返回包含了数据的一个对象：`self.scale`是转换原始分数到比例分数（或者相反）的`Interpolator`；`scores`是（得分，频率）对的列表。

`score_pmf`是缩放分数的Pmf对象。`self.raw`是原始分数的Pmf对象。`self.prior`是`p_correct`的Pmf对象。

图12-1显示了`p_correct`的先验分布。这种分布近似于高斯分布，

但是它在两端很平。在设计上，SAT强调了平均值两个标准差内的测试者得分，而忽略超出该范围的部分。

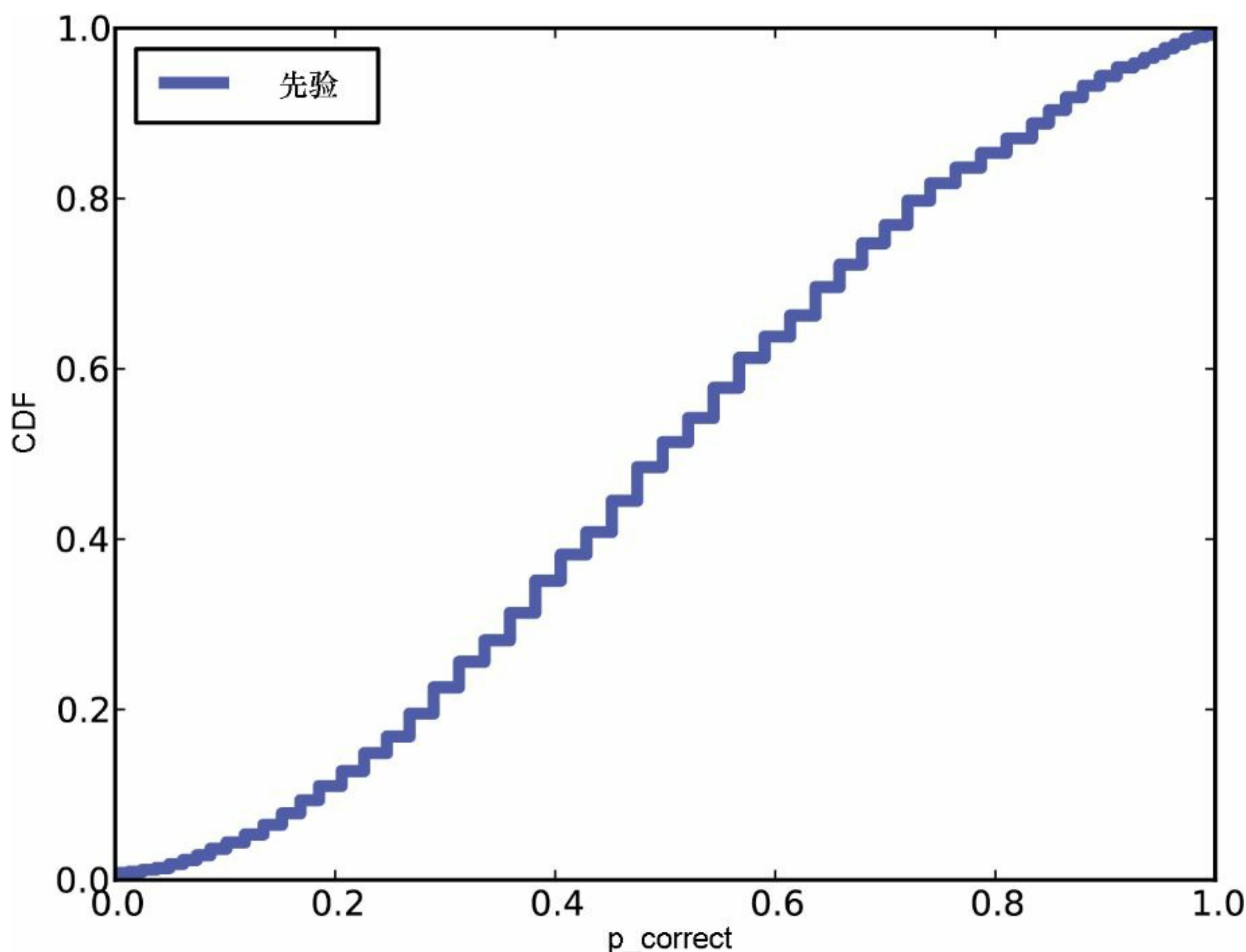


图12-1　SAT考试考生p_correct的先验分布

对于每一个参加测试者，我定义了一个名为`Sat`的Suite对象，代表`p_correct`的分布。定义如下：

```
class Sat(thinkbayes.Suite):

    def __init__(self, exam, score):
        thinkbayes.Suite.__init__(self)
        self.exam = exam
        self.score = score

        # start with the prior distribution
        for p_correct, prob in exam.prior.Items():
            self.Set(p_correct, prob)
```

```
        # update based on an exam score
        self.Update(score)
```

__init__ 接收一个Exam对象和比例分数。它创建一个先验的副本，再根据考试成绩更新这个副本。

像往常一样，我们从Suite 继承Update，再改写Likelihood：

```
def Likelihood(self, data, hypo):
       p_correct = hypo
       score = data

       k = self.exam.Reverse(score)
       n = self.exam.max_score
       like = thinkbayes.EvalBinomialPmf(k, n, p_correct)
       return like
```

hypo 是p_correct 的一个假设值，data 为比例分数。

为了简单起见，原始分数就是正确答案的数量，忽略错误答案的罚分(每题1/4分)。这样的话，似然度由计算从n 个试题中得到k 个正确回答的概率的二项分布给出。

12.4 后验

图12-2显示了在爱丽丝和鲍勃分数基础上得到的p_correct 的后验分布。我们可以看到，它们重叠在一起，有可能鲍勃的p_correct 更高，但似乎不可能。

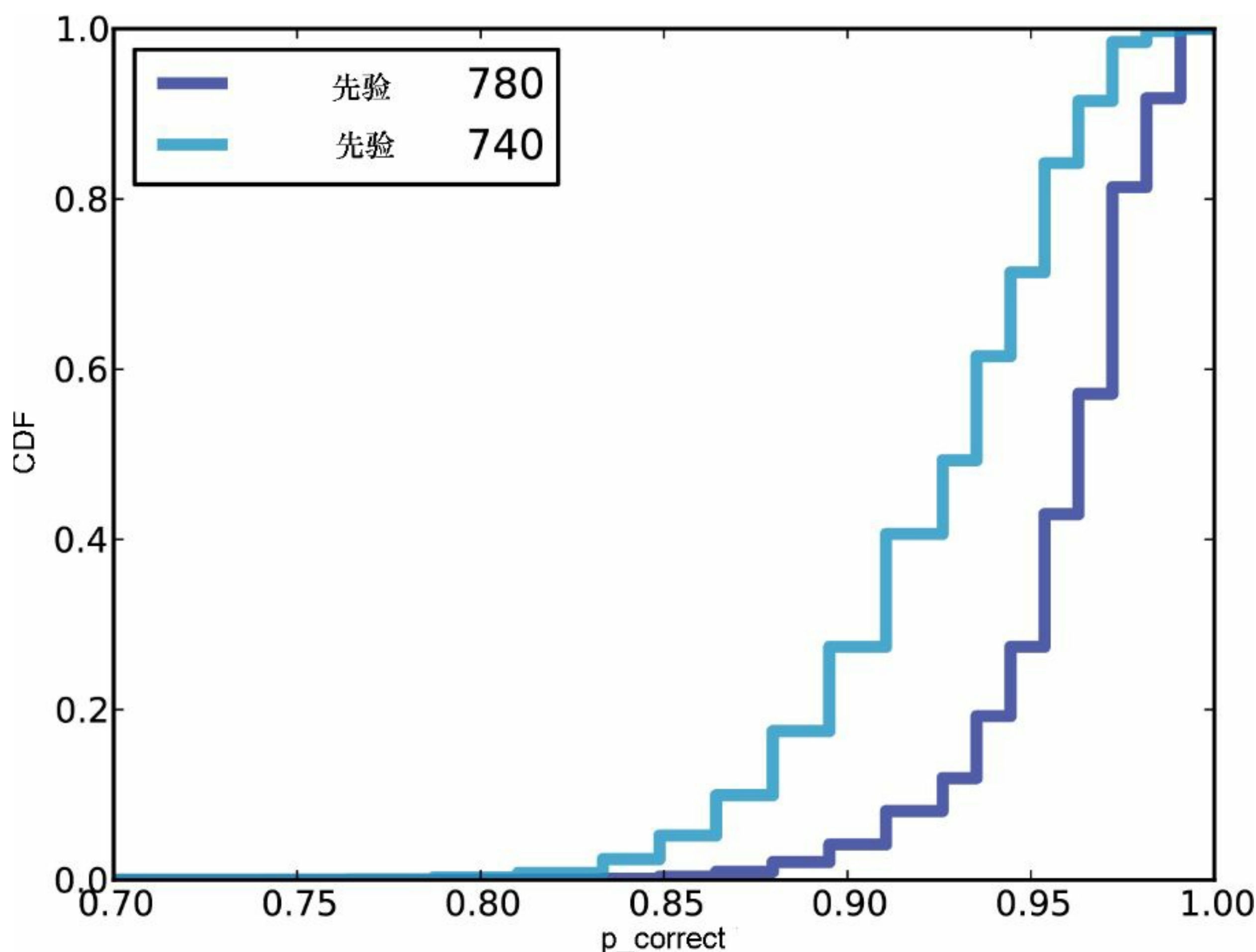


图12-2　爱丽丝和鲍勃的p_correct的后验分布

我们回到原来的问题，“有多强的证据表明，爱丽丝比鲍勃准备得更好？”我们可以用`p_correct`的后验分布来回答这个问题。

要以贝叶斯假设检验的形式定义问题，我定义了两个假设：

- A：`p_correct`上，爱丽丝高于鲍勃。
- B：`p_correct`上，鲍勃高于爱丽丝。

要计算A的似然度，可以从后验分布中枚举所有数值对，再累加得到所有爱丽丝高于鲍勃的`p_correct`概率总和，我们已经有一个函数`thinkbayes.PmfProbGreater`来实现它。

因此，我们可以定义一个计算A和B的后验概率的Suite对象：

```
class TopLevel(thinkbayes.Suite):
```

```
    def Update(self, data):
        a_sat, b_sat = data

        a_like = thinkbayes.PmfProbGreater(a_sat, b_sat)
        b_like = thinkbayes.PmfProbLess(a_sat, b_sat)
        c_like = thinkbayes.PmfProbEqual(a_sat, b_sat)

        a_like += c_like / 2
        b_like += c_like / 2

        self.Mult('A', a_like)
        self.Mult('B', b_like)

        self.Normalize()
```

通常，当我们定义一个新的Suite对象时，会继承Update，并（根据模型）实现Likelihood。本例中，则需要重写Update，因为这样能更容易同时评价两种假设的似然度。

传递到Update 的数据是表示了p_correct 后验分布的Sat 对象。

a_like 是爱丽丝较高的p_correct 的总概率；b_like 是鲍勃较高的p_correct 的总概率。

c_like 是两者“相等”的概率，在使用一些离散值为p_correct 建模的情况下，这种相等性就是人为的了。因为，如果我们使用更多的值，c_like 会较小，在极端情况下，如果p_correct 是连续的，c_like 为零。所以我将 c_like 作为一种舍入误差并且在a_like 和b_like 之间的匀去。

下面的代码创建了TopLevel 并更新它：

```
exam = Exam()
    a_sat = Sat(exam, 780)
    b_sat = Sat(exam, 740)

    top = TopLevel('AB')
    top.Update((a_sat, b_sat))
    top.Print()
```

A 的似然度是0.79，*B* 的似然度为0.21。似然比(或贝叶斯因子)为3.8，这意味着这些考试成绩的证据表明，在SAT成绩上，爱丽丝优于鲍勃。看到考试成绩前，如果我们相信*A* 和*B* 可能相等，那么在看到成绩之后，我们应该相信*A* 的概率是79％，这意味着仍然有21％的可能，鲍勃实际准备得更好。

12.5　一个更好的模型

请记住，我们迄今所做的分析都是基于所有SAT问题是同等难度的前提下的。实际上，有些题比其他题要容易，这意味着爱丽丝和鲍勃之间的差异可能会更小。

但是这有多大的建模误差？如果误差小，我们可以得出第一个模型（同等难度）足够好的结论。如果误差大，就需要一个更好的模型。

在接下来的几节中，我们开发了一个更好的模型，并将会发现（剧透一下）建模误差就是小的。所以，如果你满意简化模型，可以跳过这些内容直接到下一章。如果你想了解更真实的模型设计过程，接着往下看……

- 假设每个测试者具有一定程度的答题效率`efficacy`，答题效率衡量其回答SAT问题的能力。
- 假定每个问题有不同水平的难度`difficulty`。
- 最后，假设一个测试者正确回答问题的机会与答题效率`efficacy`和难度水平`difficulty`相关，并由下面函数决定：

```
def ProbCorrect(efficacy, difficulty, a=1):
    return 1 / (1 + math.exp(-a * (efficacy - difficulty)))
```

此函数是项目响应理论 的曲线的一个简化版本，你可以参考http://en.wikipedia.org/wiki/Item_response_theory 。答题效率和难度水平基于同一刻度水平，得到正确答案的概率就只取决于它们之间的差异。

当`efficacy`和`difficulty`相同时，正确的回答问题的概率为50%。当`efficacy`增加，概率接近100%，当它降低（或者`diffiulty`

增加），概率接近0%。

已知答题者在效率上的分布和所有问题难度的分布，我们就可以计算原始分数的预期分布。我们将通过两步完成。首先，对于已知efficacy 的某个答题者，我们将计算原始分数的分布如下：

```
def PmfCorrect(efficacy, difficulties):
    pmf0 = thinkbayes.Pmf([0])

    ps = [ProbCorrect(efficacy, diff) for diff in difficulties]
    pmfs = [BinaryPmf(p) for p in ps]
    dist = sum(pmfs, pmf0)
    return dist
```

difficulties 是难度列表，每一个试题对应一个难度值。ps 为概率的列表，pmfs 是这两个值的Pmf对象列表。下面是对应的创建函数：

```
def BinaryPmf(p):
    pmf = thinkbayes.Pmf()
    pmf.Set(1, p)
    pmf.Set(0, 1-p)
    return pmf
```

dist 是这些Pmfs的总和。还记得40页当我们添加Pmf对象时的“加数”，结果是总和的分布。为了使用Python的sum 函数来累加Pmfs，我们需要提供pmf0 作为Pmfs的标识，所以pmf + pmf0 就等于pmf 了。

如果知道答题者的效率，我们可以计算他们原始分数的分布。对于一群有不同的答题效率的人，所产生的原始分数的分布是混合的。下面是计算混合分布的代码：

```
# class Exam:

    def MakeRawScoreDist(self, efficacies):
        pmfs = thinkbayes.Pmf()
        for efficacy, prob in efficacies.Items():
            scores = PmfCorrect(efficacy, self.difficulties)
            pmfs.Set(scores, prob)
        mix = thinkbayes.MakeMixture(pmfs)
        return mix
```

MakeRawScoreDist 接收efficacies，这是一个表示所有答题者效率分布的Pmf对象。我假设它是均值为0，标准偏差1.5的高斯分布。这一假设相当主观。得到一个问题的正确的概率取决于答题效率和试题难度的区别，所以我们可以选择效率的单位，再校准相应的题目难度的单位。

pmfs 是包含每一级别答题效率Pmf的一个元Pmf，并映射到同一级别的测试者上。MakeMixture 接收元Pmf并计算混合的分布(参见第45页上的“混合分布”)。

12.6 校准

如果我们难度的分布情况，我们就可以使用MakeRawScoreDist 计算原始分数的分布。但对于我们来说，问题是类似的其他方法：有原始分数的分布，要推断难度的分布。 假设难度的分布是带有参数center 和width 的均匀分布，MakeDifficulties 可以得到这些参数下试题难度的列表。

```
def MakeDifficulties(center, width, n):
    low, high = center-width, center+width
    return numpy.linspace(low, high, n)
```

通过尝试了几个组合，我发现，center = −0.05和width = 1.8得到的原始分数分布类似于实际数据，如图12-3所示。

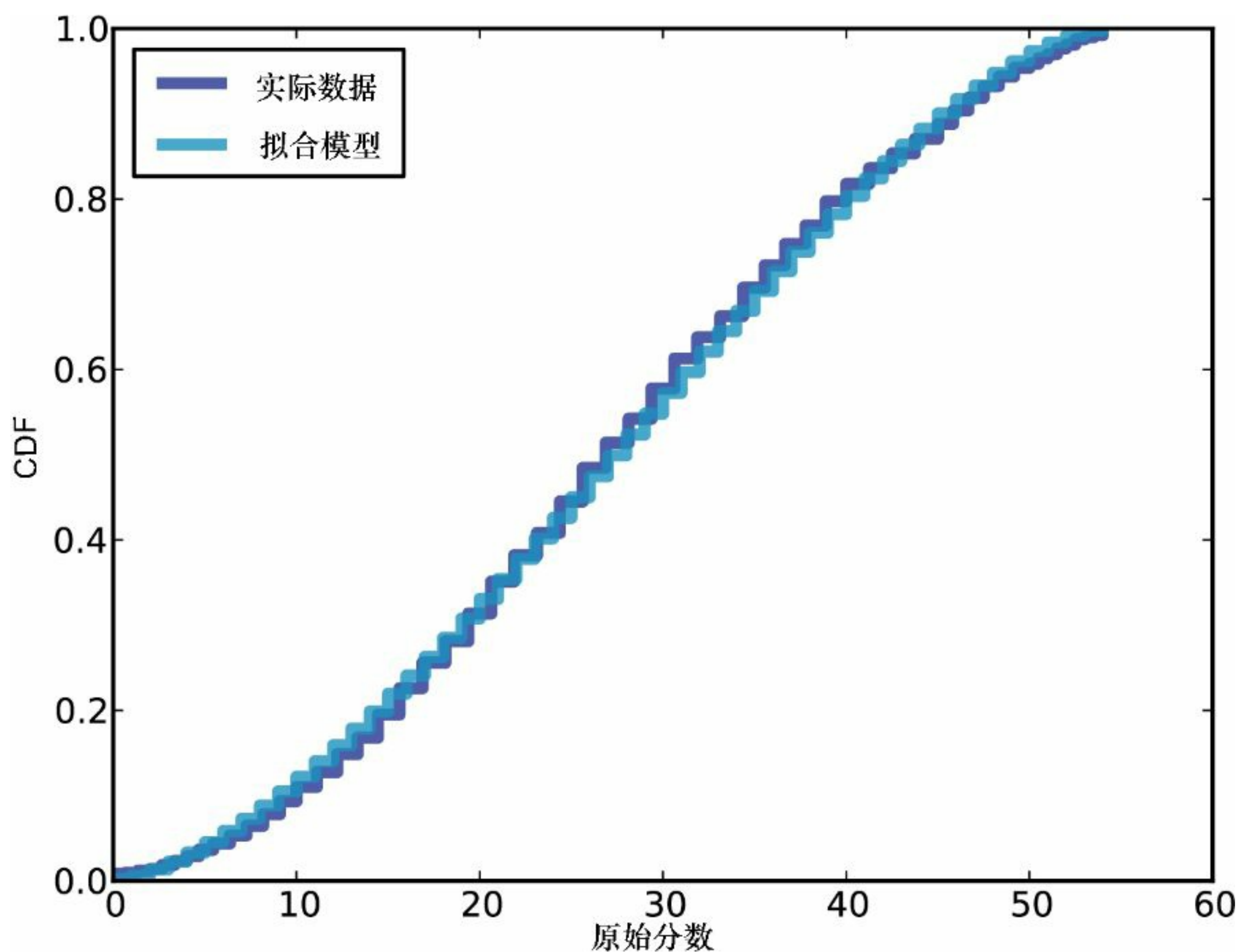


图12-3　原始分数的实际分布和一个拟合它的模型

因此，假设难度的分布是均匀分布，其范围大约是−1.85至1.75，答题效率是均值为0，标准偏差1.5的高斯分布。

下表显示了不同效率级别的测试者`ProbCorrect`的范围：

答题效率（**Efficacy**）	难度（**Difficulty**）		
	−1.85	−0.05	1.75
3.00	0.99	0.95	0.78
1.50	0.97	0.82	0.44
0.00	0.86	0.51	0.15
−1.50	0.59	0.19	0.04
−3.00	0.24	0.05	0.01

效率为3的答题者(两个标准差高于平均值)有99%的机会答对最简单的问题，78%的机会答对最难的问题。在区间的另一端，低于均值两个标准偏差的答题者，只有24%的机会答对最简单的问题。

12.7　效率的后验分布

现在，该模型已经被校准了，我们可以为爱丽丝和鲍勃计算答题效率的后验分布。下面是一个使用该模型的Sat类的新版本：

```
class Sat2(thinkbayes.Suite):

    def __init__(self, exam, score):
        self.exam = exam
        self.score = score

        # start with the Gaussian prior
        efficacies = thinkbayes.MakeGaussianPmf(0, 1.5, 3)
        thinkbayes.Suite.__init__(self, efficacies)

        # update based on an exam score
        self.Update(score)
```

`Update` 调用`Likelihood`，这计算出已知SAT得分时，答题者假设效率水平的似然度。

```
def Likelihood(self, data, hypo):
     efficacy = hypo
     score = data
     raw = self.exam.Reverse(score)

     pmf = self.exam.PmfCorrect(efficacy)
     like = pmf.Prob(raw)
     return like
```

`pmf` 是已知效率的答题者得到的原始得分的分布，`like` 是观察到分数的概率。

图12-4显示了爱丽丝和鲍勃效率的后验分布。正如预期的那样，爱丽丝的分布位置更远，靠近右边，但同样有一些重叠部分。

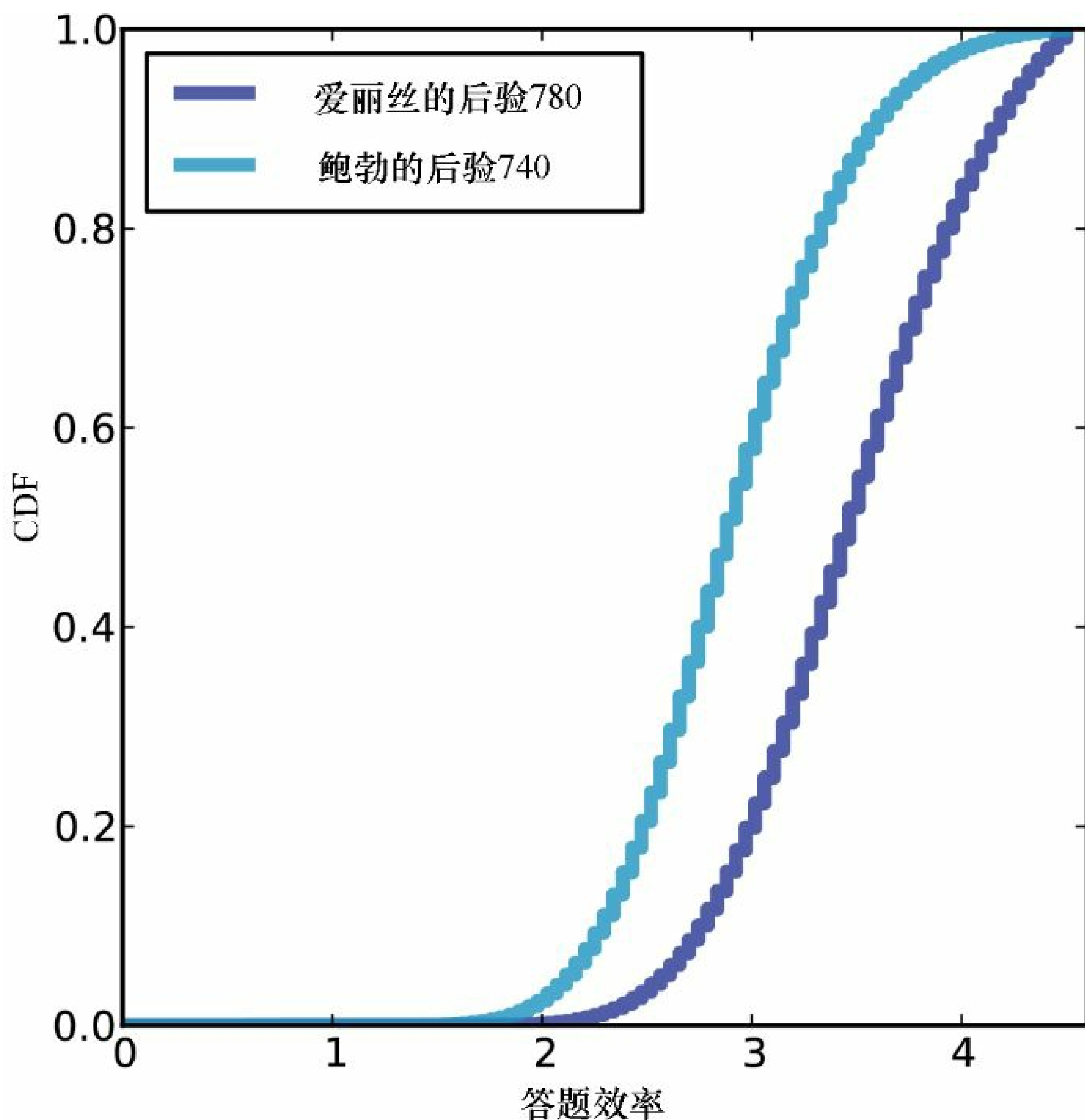


图12-4　爱丽丝和鲍勃答题效率的后验分布

再次使用`TopLevel`，我们比较假设*A*（爱丽丝效率更高的假设）和假设*B*（鲍勃效率更高的假设）。似然比为3.4，比我们从简化模型（3.8）得到的小些。因此，这个模型表明，数据的证据支持假设*A*，但弱于之前的估计。

如果我们的先验是*A*和*B*等可能，那么参考这个证据，我们会给假设*A* 77％的后验概率，另外有23％的可能，鲍勃的效率更高。

12.8　预测分布

到目前为止我们所做的分析估计了爱丽丝和鲍勃的效率，但由于效率是无法直接观察到的所以难以验证结果。

为了让模型有预测的能力，可以用它来回答一个相关的问题："如果爱丽丝和鲍勃再进行一次SAT数学测试，爱丽丝比鲍勃得分高的可能性是多少？"

我们将通两个步骤回答这个问题：

- 使用效率的后验分布来生成每个测试接受者原始得分的后验分布。
- 比较这两个预测分布，计算爱丽丝得到更高分数的概率。

我们已经有了大部分需要的代码。为了计算预测分布，可以再次使用MakeRawScore-Dist：

```
exam = Exam()
a_sat = Sat(exam, 780)
b_sat = Sat(exam, 740)

a_pred = exam.MakeRawScoreDist(a_sat)
b_pred = exam.MakeRawScoreDist(b_sat)
```

接着，我们可以得到在第二次测验中爱丽丝比鲍勃分高，鲍勃分高，或者他们分数相同的可能性：

```
    a_like = thinkbayes.PmfProbGreater(a_pred，b_pred)
    b_like = thinkbayes.PmfProbLess(a_pred，b_pred)
    c_like = thinkbayes.PmfProbEqual(a_pred，b_pred)
```

爱丽丝在第二次测验中得分更高的概率是63％，这意味着鲍勃更高分的概率是37％。

请注意，我们对爱丽丝的效率更有信心，高于下一次测验的信心。爱丽丝的效率较高的后验赔率是3:1，但在下一个测试中，爱丽丝更好的赔率只有2:1。

12.9　讨论

我们以问题“爱丽丝比鲍勃准备更充分的证据有多强”为本章的开始，这个问题表面上听起来像我们想测试两个假设：要么爱丽丝，要么鲍勃准备得更好。

但为了计算这些假设的似然度，我们必须解决一个估计问题。对于每个参加测试者，我们必须找到`p_correct`或`efficacy`的后验分布。

这样的值称为干扰参数，因为我们实际上不关心它们是什么，但为了回答所关心的问题必须估计这些量。

本章中我们实现可视化分析结果的方法是绘制这些参数的空间分布。`thinkbayes.MakeJoint`接收两个Pmfs对象，计算它们的联合分布，并返回每个可能的值和概率对的概率密度函数。

```
def MakeJoint(pmf1, pmf2):
    joint = Joint()
    for v1, p1 in pmf1.Items():
        for v2, p2 in pmf2.Items():
            joint.Set((v1, v2), p1 * p2)
    return joint
```

此函数假设两个分布是独立的。

图12-5显示了`p_correct`（爱丽丝和鲍勃）的联合后验分布。空间中的对角线表示爱丽丝和鲍勃的`p_correct`相同的情况。在这条线的右边，说明爱丽丝准备得更好；左边，说明鲍勃准备得更好。

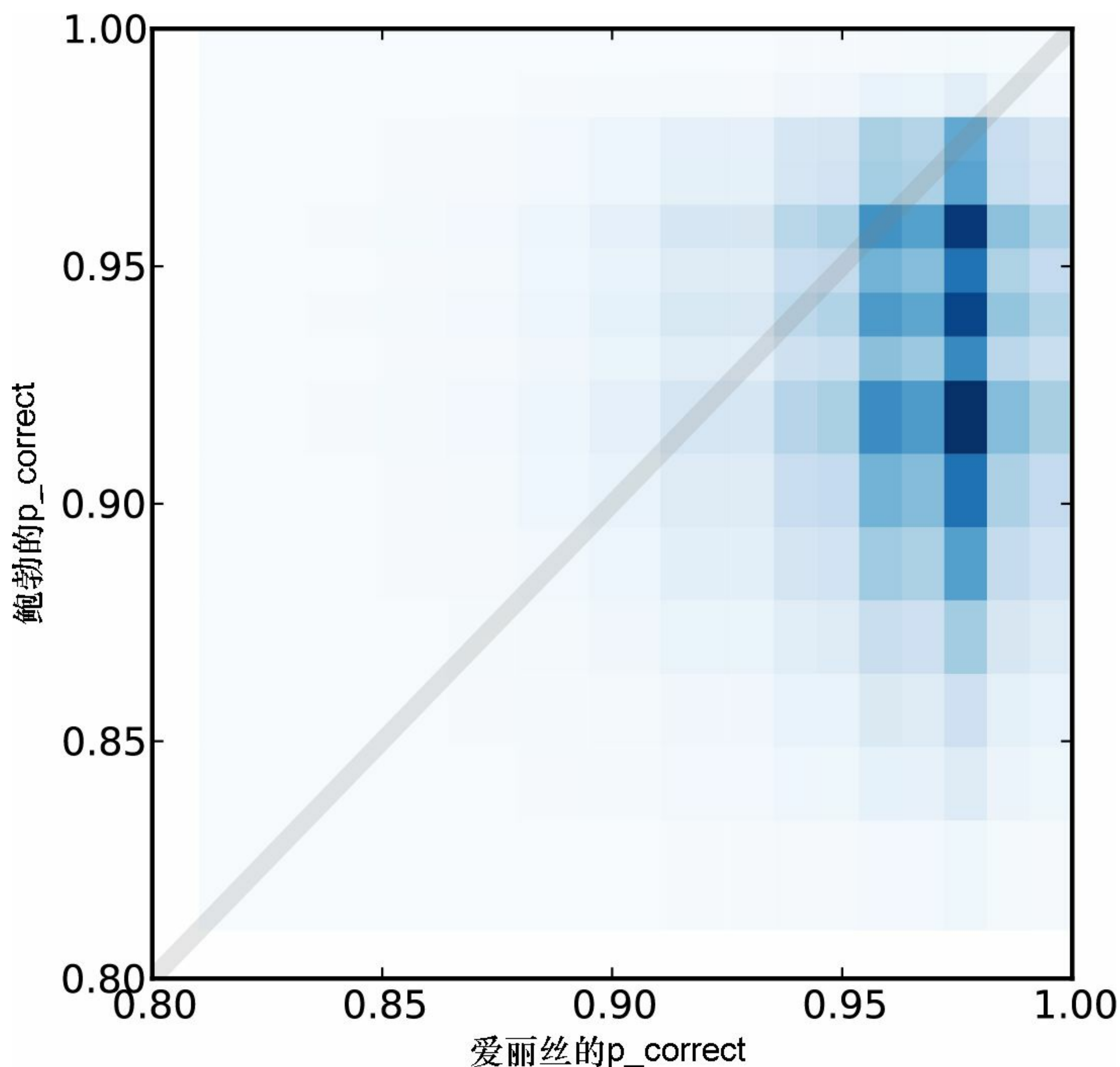


图12-5　爱丽丝和鲍勃p_correct的联合后验分布

在`TopLevel.Update`中，当计算*A*和*B*的似然性时，我们累加了这条线两侧的概率质量。对于落在该行的单元格，我们累加*A*和*B*之间的总质量，并把它匀到*A*和*B*中。

我们本章中使用的过程——为了计算两个互斥假设的似然度而估计干扰参数——是一种常见的解决类似问题的贝叶斯方法。

第13章　模拟

在本章中，我描述了一个肾肿瘤患者所提问题的解决方案。我认为这个问题对于患者和进行医治的医生来说都是重要和有关系的。

我认为它很有趣，因为这虽然是一个贝叶斯问题，但是使用贝叶斯的方式却是隐含的。我列出了解法和代码，在本章结尾，我将解释贝叶斯的部分。

如果你想了解更多的技术细节，可以在 *http://arxiv.org/abs/1203.6890* 阅读我有关这项工作的论文。

13.1　肾肿瘤的问题

我是在线统计论坛 *http://reddit.com/r/statistics* 的忠实用户，也偶尔贡献些内容。2011年11月，我在那儿读到了以下消息：

“我现在处于第Ⅳ期肾癌，想确定癌症是否是在我从部队退役以前就形成的……提供退伍和确诊的日期是否可以确定有 50/50 的可能我是这么得病的？是否有可能确定我在退伍日期时患病的概率是多少？确诊时，我的肿瘤为15.5×15厘米，Ⅱ级。”

我联系了消息的作者，并获得了更多的信息，我了解到，如果肿瘤“可能而非不是”[①] 是在部队服役期间形成的，退伍军人可以得到补偿是不同的（除其他因素外）。

因为肾肿瘤生长缓慢，通常也没有什么引发的症状，可以有时并不进行治疗。但医生还是会观察肿瘤，并比较同一病人未处理的肿瘤在不同时间的生长速率。有几篇论文报导了这些生长率。

我从Zhang [②] 的论文中找到了一些数据，并且联系了作者看能否得到原始数据。不过他们以患者隐私的原则回绝了。不过我还是能够绘制出他们的报告数据图形，再按规则进行测量来提取出我需要的数据。

他们以倍增时间增长率倒数（RDT）的形式报告了增长率，即以倍

增体/每年的形式。因此，*RDT* =1表示肿瘤每年增长双倍体积；*RDT* =2表示它每年四倍；*RDT* =-1，表示一半。图13-1显示了53例患者RDT的分布。

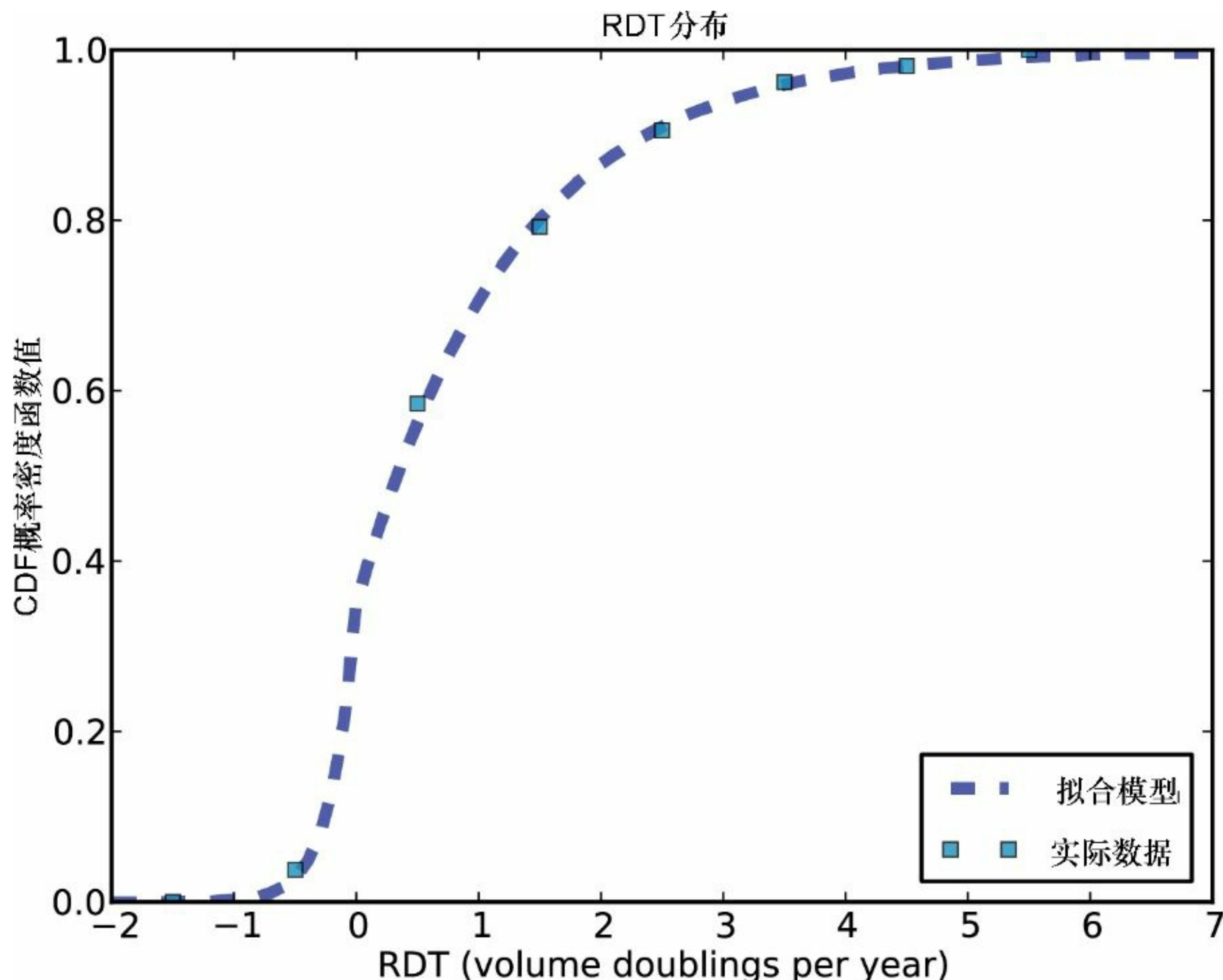


图13-1　RDT（每年倍增体积）的CDF

方块是论文上的数据点；虚线是我就数据拟合而成的模型。该线条的正值部分和指数分布拟合得很好，所以我用了两个混合的指数函数。

13.2　一个简化模型

在尝试更有挑战性的东西之前，采用简化模型一般是不错的。对于手头的一些问题简化模型有时就足够了，而且如果不是，你还可以用简化模型来验证更复杂的模型。

我的简化模型是：认为肿瘤的生长具有恒定的倍增时间，而且肿瘤是三维的，即如果每一维长度测量值翻倍，体积就是增长八倍（2×2×2＝8）。

我从案例的合作者解到，他从军队退伍到诊断日是3291天（约9年）。所以，首先要计算的就是“如果肿瘤以中位速率增长，那么它在退伍日时有多大？”

体积倍增时间的中位数，从Zhang的报告中看约为811天。假设一个三维几何体，长度值的倍增时间为该值的3倍。

```
# time between discharge and diagnosis, in days
interval = 3291.0

# doubling time in linear measure is doubling time in volume * 3
dt = 811.0 * 3

# number of doublings since discharge
doublings = interval / dt

# how big was the tumor at time of discharge (diameter in cm)
d1 = 15.5
d0 = d1 / 2.0 ** doublings
```

你可以从http://thinkbayes.com/kidney.py 下载本章代码。更多信息，请参见前言的“代码指南”。

结果`d0` 约6厘米。因此，如果这个肿瘤是在退伍日期后形成的，它必须以大幅超过平均速度的速度增长。因此，我可以断言肿瘤是“可能而非不是”在退伍前形成的。

此外，我计算的增长率能暗示肿瘤是否是退伍前形成的。如果假设其初始大小为0.1厘米，我们可以计算出达到15.5厘米最终尺寸的倍增量：

```
# assume an initial linear measure of 0.1 cm
d0 = 0.1
d1 = 15.5

# how many doublings would it take to get from d0 to d1
```

```
doublings = log2(d1 / d0)

# what linear doubling time does that imply?
dt = interval / doublings

# compute the volumetric doubling time and RDT
vdt = dt / 3
rdt = 365 / vdt
```

dt 是线性的倍增时间，所以vdt是体积的倍增时间，rdt 是倒数倍增时间。

倍增量以线性长度衡量是7.3，这意味着RDT是2.4。在Zhang等人的数据中，只有20%的肿瘤在观察期中增长这么快。

所以，再一次，我得出的结论是“较有可能”肿瘤形成于退伍前。

这些计算都足以回答前面的那个问题，我代表合作者给退伍军人福利处VBA写了一封信，解释我的结论，后来我还将结果告诉了我的一个肿瘤科医生朋友。他对Zhang等人观察到的肿瘤增长速度和生长年龄感到惊讶。他指出对于研究人员和医生，这一结果都会引起关注。

但是为了让模型更有用，我想要找到一个更普遍的肿瘤年龄和大小之间的模型。

13.3 更普遍的模型

已知肿瘤在诊断时的大小，如果知道肿瘤在某一个给定时间前形成的概率，即肿瘤年龄（生长时间）的分布是最有用的。

为了得出结论，我运行了模拟肿瘤生长的程序，得到在已知生长年龄的条件时，肿瘤大小的分布。然后我们可以用贝叶斯方法得到已知肿瘤尺寸条件时的年龄分布。

模拟以一个小肿瘤开始，并以下面这些步骤运行：

1．从RDT的分布中选取一个增长率；

2．在每个区间的结尾计算肿瘤的尺寸；

3．记录在每个时间间隔里肿瘤的尺寸；

4．重复，直到肿瘤超过最大的相应尺寸。

对于初始尺寸，我选取了0.3厘米，因为比这小的肿瘤不太可能有侵入性以及快速增长所需的血液供应（见 *http://en.wikipedia.org/wiki/Carcinoma_in_situ* ）。

我选择了245天（约8个月）的区间，因为这是数据源中测量对象的平均时间。

最大尺寸我选择了20厘米，在数据源中，观察肿瘤的尺寸范围是1.0～12.0厘米，所以可以推断这超出了观测范围的两端，但不多，而且不太可能对结果有显著影响。

仿真基于一个较大的简化：增长率设置为在每个区间内是独立互不影响的，因此它不依赖于年龄、大小，或前一个区间的生长速率。

在第135页的“序列相关性”中，我回顾评估了这些假设并考虑了更加详细的模型。不过我们首先来看一些例子。

图13-2显示了模拟的肿瘤尺寸函数（图形），以年龄作为变量。10厘米处的虚线显示了肿瘤在该尺寸的年龄范围：增长最快的肿瘤为8年；最慢的超过35年。

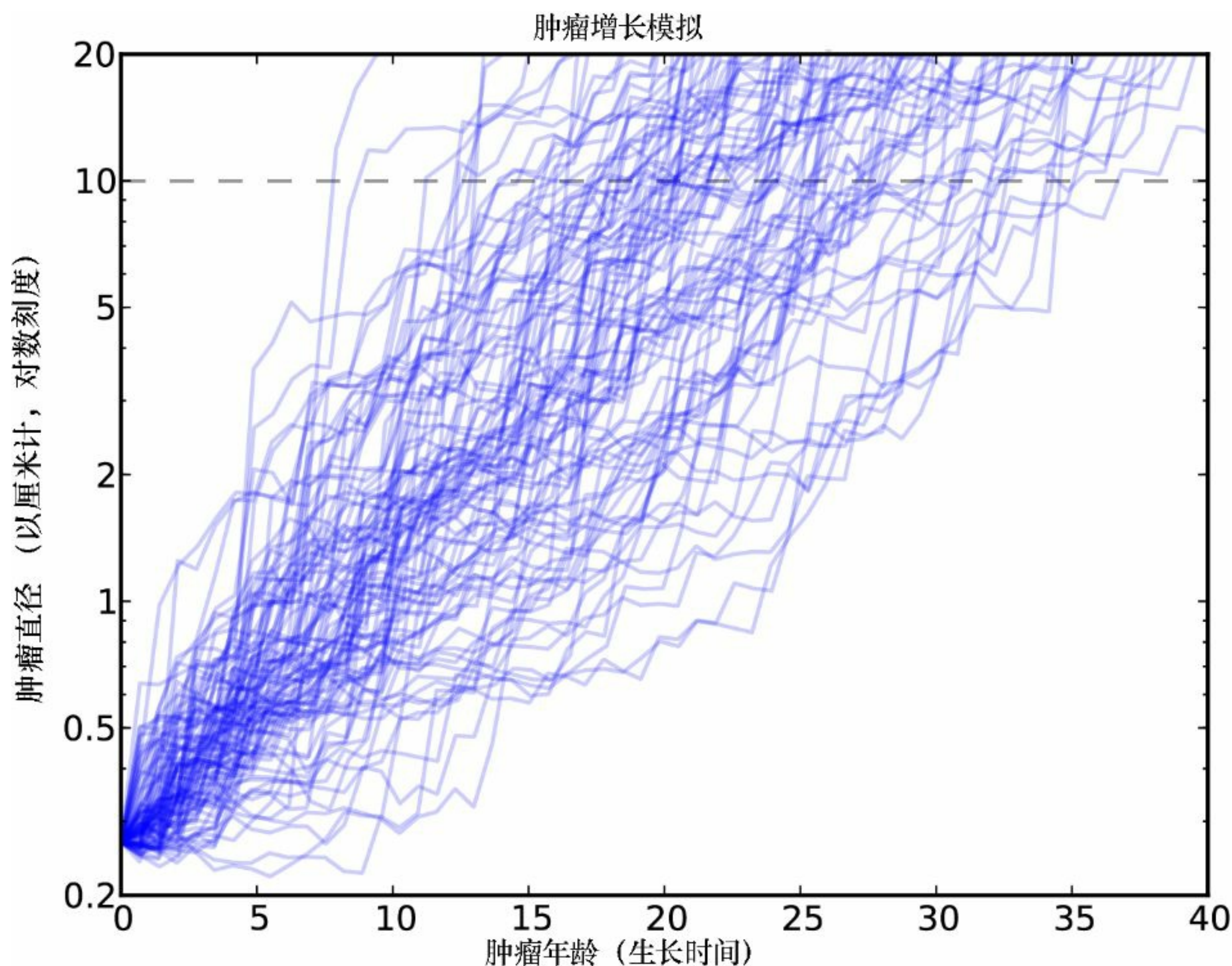


图13-2　肿瘤增长的模拟时间和大小

我用线性测度展示出结果，但计算实际上是依据体积进行的。同样，为了从一个转换到另一个，我以给定的直径来表示球体的体积。

13.4　实现

下面是这一模拟的核心代码：

```
def MakeSequence(rdt_seq, v0=0.01, interval=0.67, vmax=Volume(20.0)):
    seq = v0,
    age = 0

    for rdt in rdt_seq:
        age += interval
        final, seq = ExtendSequence(age, seq, rdt, interval)
```

```
        if final > vmax:
            break

    return seq
```

rdt_seq 是从增长率 CDF 产生随机数的迭代器。v0 是以毫升（mL）表示的初始体积。interval 是以年计的时间间隔。vmax 是对应20厘米长度的最终体积。

Volume 把长度测量的厘米（cm）数转化为毫升（mL）体积，基于该肿瘤是一个简化球体这个前提条件：

```
def Volume(diameter, factor=4*math.pi/3):
    return factor * (diameter/2.0)**3
```

ExtendSequence 在时间间隔结束时计算肿瘤的体积。

```
def ExtendSequence(age, seq, rdt, interval):
    initial = seq[-1]
    doublings = rdt * interval
    final = initial * 2**doublings
    new_seq = seq + (final,)
    cache.Add(age, new_seq, rdt)

    return final, new_seq
```

age 是肿瘤在间隔区间结束时的生长年龄。seq 是一个包含当前体积的元组。rdt 是间隔期间的增长率，每年倍增。interval 是以年计的时间步长。

返回值final 是肿瘤在时间间隔结束时的体积，而new_seq 是一个新的含有seq 加上新算出的体积final 的元组。

Cache.Add 在每个时间间隔的末尾记录每个肿瘤的年龄和尺寸，下面会进行解释。

13.5　缓存联合分布

以下是cache的功能。

```
class Cache(object):

    def __init__(self):
        self.joint = thinkbayes.Joint()
```

joint 是一个记录每个年龄-尺寸对频率的联合Pmf 对象，所以它近似于年龄和大小的联合分布。

在每个模拟间隔结束时，ExtendSequence 调用Add：

```
# class Cache

    def Add(self, age, seq):
        final = seq[-1]
        cm = Diameter(final)
        bucket = round(CmToBucket(cm))
        self.joint.Incr((age, bucket))
```

同样，age 为肿瘤的年龄，seq 是肿瘤目前体积的值序列。

添加新数据到联合分布前，我们用Diameter 将体积转换到直径，以厘米为单位：

```
def Diameter(volume, factor=3/math.pi/4, exp=1/3.0):
    return 2 * (factor * volume) ** exp
```

CmToBucket 将厘米转换到一个离散的量bucket：

```
def CmToBucket(x, factor=10):
    return factor * math.log(x)
```

buckets等间隔分布于对数标度。利用factor = 10得出一个合理的buckets数量；例如，1厘米映射到bucket 0，10厘米映射到bucket 23 ③ 。

运行模拟后，我们就可以绘制联合分布的伪彩图，其中每个单元格

代表在给定的大小—年龄对观察到的肿瘤的数目。

图13-3显示了1000次模拟后的联合分布。

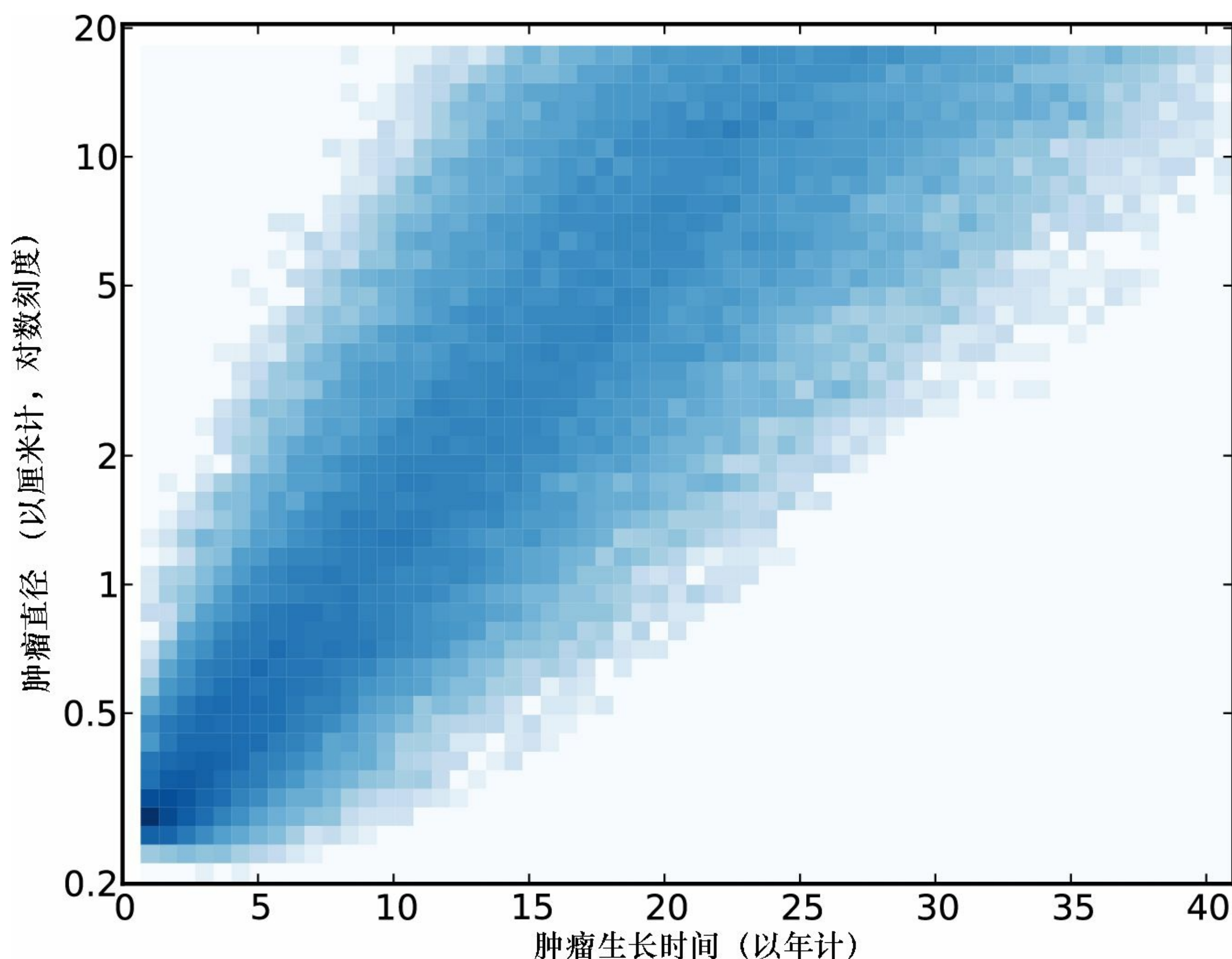


图13-3　肿瘤大小和年龄的联合分布

13.6　条件分布

通过从联合分布取垂直切片，我们可以得到对于任何给定年龄的尺寸大小分布。通过做一份水平切片，我们可以得到给定尺寸的年龄分布。

下面是对于一个给定的大小读取联合分布，并建立条件分布的代码。

```
# class Cache

    def ConditionalCdf(self, bucket):
        pmf = self.joint.Conditional(0, 1, bucket)
        cdf = pmf.MakeCdf()
```

```
        return cdf
```

bucket 是对应于肿瘤大小的整数值。Joint.Conditional 计算给定bucket 值下年龄的PMF。其结果是给定bucket 下年龄的CDF。

图13-4显示了这样几个在不同大小下的CDF。总结这些分布，我们可以计算出不同大小函数下的百分位数。

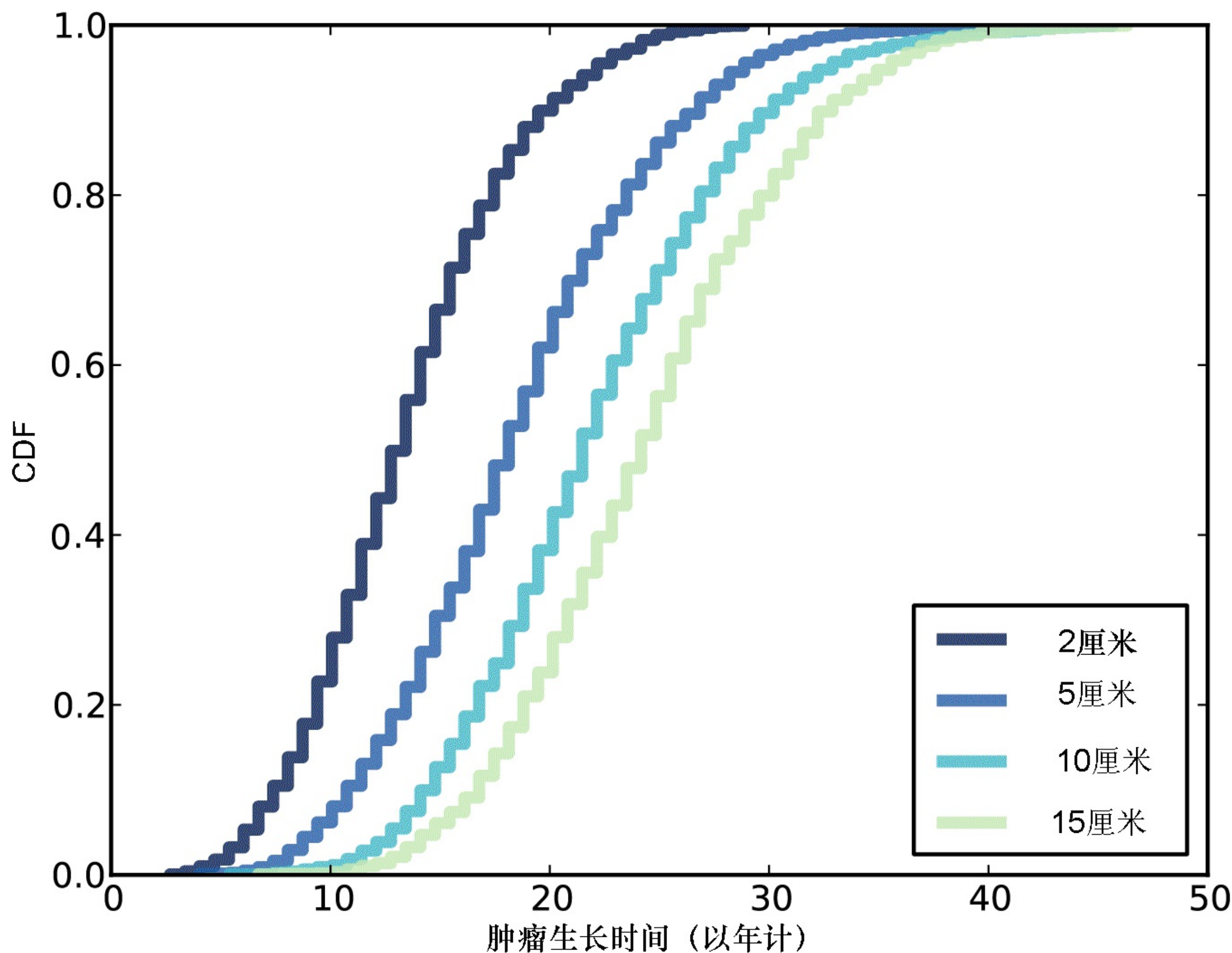


图13-4　给定大小条件下，肿瘤生长年龄的分布

```
percentiles = [95, 75, 50, 25, 5]

for bucket in cache.GetBuckets():
    cdf = ConditionalCdf(bucket)
    ps = [cdf.Percentile(p) for p in percentiles]
```

图13-5显示了每个大小区间的这些百分位数。数据点从估计的联合分布中计算得到。在该模型中大小和时间是离散的，这产生了数值误差，所以我也展示了就每个百分位数序列的最小二乘拟合曲线。

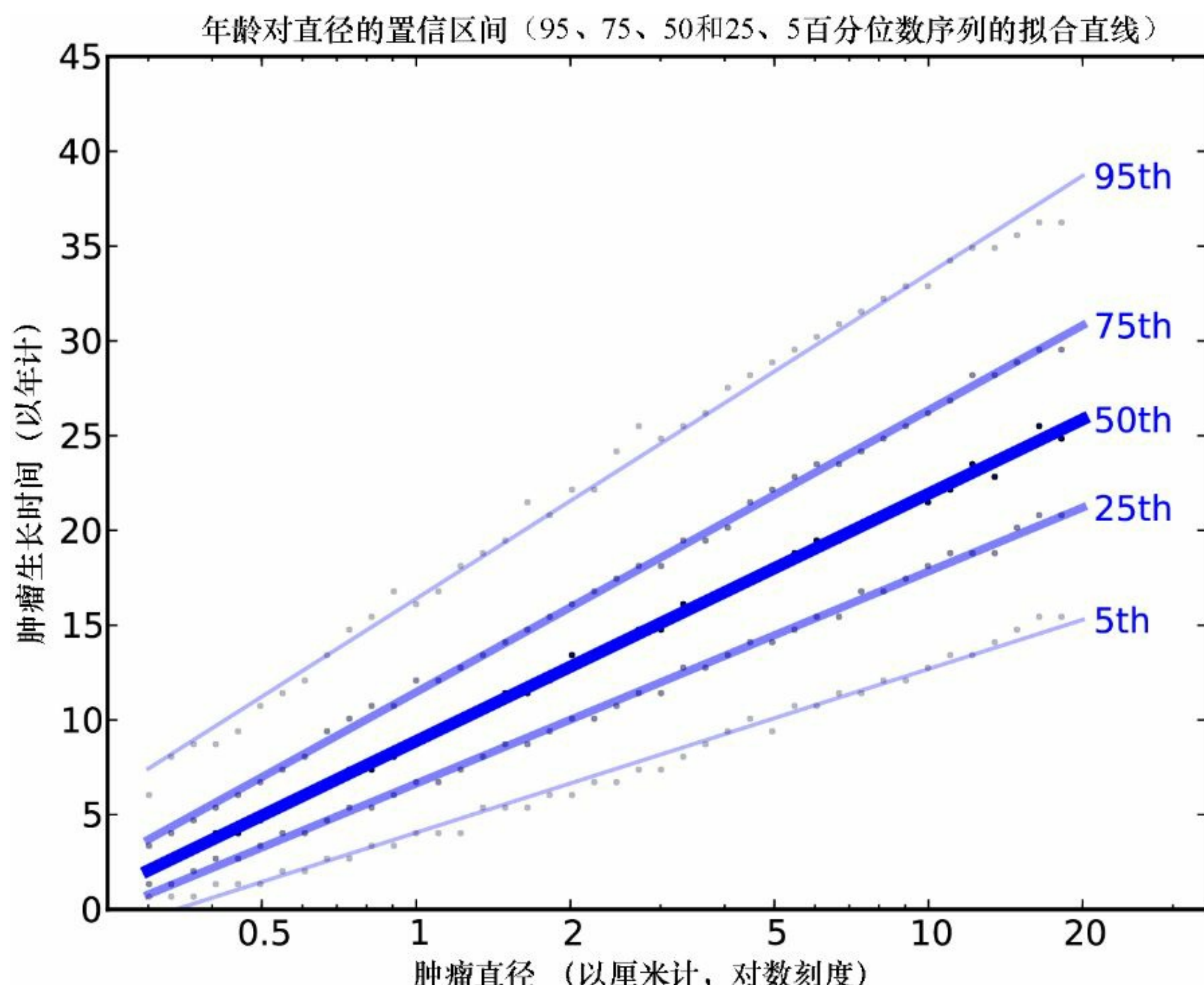


图13-5　肿瘤的年龄作为大小的函数的百分位数

13.7　序列相关性

到目前为止，得到的结果都是基于多项建模决定的；让我们回顾一下这些建模，看看哪些是误差最有可能的来源。

- 要转换线性测量到体积，我们假设肿瘤近似于球形。这个假设对于几厘米的肿瘤很合适，对非常大的肿瘤却不行。
- 在模拟过程中，增长率的分布是基于一个我们选择的和Zhang所报导数据拟合的连续模型，数据基于53名患者。该拟合只是近似的，更重要的是，大一些的样本可能产生不同的分布。
- 生长模型没有考虑肿瘤亚型或等级。这一假设是为了与Zhang等人的结论保持一致：“不同大小、子类型和等级肾肿瘤的增长率构成

了一个广泛的范围并且实际上重叠了在一起了。”但是在大的样本量下，它们之间的差异可能会变得更明显。

- 生长速率的分布不依赖于肿瘤的大小。对于非常小和非常大的肿瘤，这一假设是不符合实际的，增长是由血液供应所限制的。

但Zhang等人观测的肿瘤大小为1-12厘米，而他们没有发现大小和增长速度之间的统计性显著关系。所以如果实际上关联存在，它至少在这个尺寸范围内很可能只是弱的(译注：表示没有关联)。

- 在仿真中，每个间隔期间的增长速率是独立于之前间隔的增长率的。实际上这是似是而非的，迅速增长过了的肿瘤更可能(在下一个间隔)继续快速增长。换句话说，增长速率可能前后相关。

其中，第一个和最后一个似乎最有问题。首先调查下前后相关性，再回过头来考虑球面几何因素。

为了模拟有相关性的肿瘤生长，我写了一个generator ④，能够从一个给定的Cdf产生一个相关系列。下面是该算法的工作原理。

1．从高斯分布生成相关值。这很容易，因为我们能以前一值为条件计算下一值的分布。

2．利用高斯CDF，将每个值转换为其累积概率。

3．通过给定Cdf，将累积概率转换为相应值。

代码如下：

```
def CorrelatedGenerator(cdf, rho):
    x = random.gauss(0, 1)
    yield Transform(x)

    sigma = math.sqrt(1 - rho**2);
    while True:
        x = random.gauss(x * rho, sigma)
        yield Transform(x)
```

`cdf` 是所需的Cdf ; `rho` 是所求的相关性因子，`x` 是高斯值；`Transform` 再把它们转换成所需的分布。

x 的第一个值是均值为0，标准偏差为1 的高斯值。对于后续值，平均值和标准偏差依赖于先前的值。给定上一个x，下一个值的平均值为x * rho，方差为1 - rho ** 2 。

Transform 从每个高斯值x 映射到一个给定Cdf的值y 。

```
def Transform(x):
    p = thinkbayes.GaussianCdf(x)
    y = cdf.Value(p)
    return y
```

GaussianCdf 计算在x 上标准高斯分布的CDF，返回累积概率。Cdf.Value 从累积概率映射到cdf 的对应值。

根据cdf 的形状，信息可能会在转换中遗失，所以实际相关性可能比rho 更低。例如，当我从rho = 0.4的增长率产生10000个值，实际相关性为0.37。但是，由于我们只是在对正确的关系量进行猜测，这就足够接近实际了。

请记住，MakeSequence 需要以一个迭代器作为参数。该接口允许以不同的生成器作为参数：

```
iterator = UncorrelatedGenerator(cdf)
seq1 = MakeSequence(iterator)

iterator = CorrelatedGenerator(cdf, rho)
seq2 = MakeSequence(iterator)
```

在这个例子中，seq1 和seq2 从同一分布取得，但seq1 中的值是不相关的，而seq2 的值以近似于rho 的系数相关。

现在，我们可以看到序列相关性对结果产生的效果。下面的表显示了一个6厘米的肿瘤的年龄百分位数，分别采用了不相关的生成器和有相关性 ρ =0.4的生成器。

序列相关性	直径（厘米）	年龄的百分位数				
		5	25	50	75	95

0.0	6.0	10.7	15.4	19.5	23.5	30.2
0.4	6.0	9.4	15.4	20.8	26.2	36.9

相关性使得增长最快的肿瘤更快，最慢的速度更慢，所以年龄的范围就越宽。不同的是它对于低百分位数是适中的，但对于第95百分位就是6年多。为了精确计算这些百分位数，我们需要一个更好的实际前后相关性的估计。

然而，这种模式就足以回答我们开始的问题：给定一个长度尺寸是15.5厘米的肿瘤，它形成了8年以上的概率是多少？

下面是代码：

```
# class Cache

    def ProbOlder(self, cm, age):
        bucket = CmToBucket(cm)
        cdf = self.ConditionalCdf(bucket)
        p = cdf.Prob(age)
        return 1-p
```

`cm` 是肿瘤的大小；`age` 是以年计的阈值。`ProbOlder` 转换大小到bucket数，得到给定bucket下的年龄的Cdf ，并计算这个年龄超过给定值的概率。

生长没有前后相关性的条件下，一个15.5厘米肿瘤的年龄是8岁以上的概率是0.999 ，几乎确切无疑了。相关为0.4的前提下，即一个更快生长的肿瘤，概率仍然是0.995。即使相关性为0.8，概率还是0.978。

误差的另一个可能来源就是假设肿瘤近似于球形。对于一个长度尺寸为15.5 ×15厘米的肿瘤，这种假设可能不合适。如果合适，就好像是说，如果这个尺寸的肿瘤是相对平坦的，应该和半径6厘米球体的肿瘤具有相同的体积。即使考虑到更小的体积和相关性0.8，年龄大于8的可能性仍然是95％。

因此，即使考虑到建模误差，这么大的肿瘤形成不到8年也是不可能的。

13.8 讨论

好了，我们一整章都没有使用贝叶斯定理或封装了贝叶斯更新的 `Suite` 类。怎么回事？

理解贝叶斯定理的一种方法是将其作为反向得到条件概率的一个算法。给定p（$B|A$），只要我们知道p（A）和p（B），就可以计算p（$A|B$）。当然只有在计算p（$B|A$）比p（$A|B$）容易的情况下，该算法才是有用的（由于某些原因）。

在这个例子中，通过运行仿真，我们可以估算已知年龄条件时尺寸的分布，或p（尺寸|年龄）。但已知尺寸时年龄的分布，即p（年龄|尺寸）是很难得到的。因此，这似乎是一个绝好的应用贝叶斯定理的机会。

我没有这么做的原因是计算效率。要估计对于任何给定尺寸的p（尺寸|年龄），你必须运行相当量的模拟。最后，你要在很大范围的尺寸区间来计算p（尺寸|年龄）。事实上，你最终将计算整个尺寸和年龄的联合分布p（尺寸，年龄）。

而一旦得到联合分布，就的确不需要贝叶斯定理了，你可以通过切片从联合分布提取p（年龄|尺寸），这已经在`ConditionalCdf`中阐述过。

所以，我们是绕过了贝叶斯，但我们与他精神同在。

---

① 此处原文是more likely than not，也可以意译为可能，考虑案例的特殊性，应该和法律严谨行文有关，所以直译。

② Zhang等．利用一系列容积CT测量法确定肾肿瘤的生长速率分布［J］．放射学2009，1（250页）137-144。

③ 译者注：bucket直译不恰当，保留原文。

④ 如果你对Python generator不熟悉，请参见 *http://wiki.python.org/moin/Generators* 。

第14章　层次化模型

14.1　盖革计数器问题

我是从汤姆·坎贝尔-里基茨那儿知道下面这个问题的，他是“最大熵”博客 *http://maximum-entropy-blog.blogspot.com* 的作者。而他是从经典的《概率论：科学的逻辑》的作者E.T.杰恩斯那儿知道这个问题的：

> 假设一个放射源，以平均每秒r 个粒子的速度向一个盖革计数器发射粒子，但计数器只能记录击中它的粒子的一部分，一个分数f，如果f 为10%而且计数器在1秒的时间内记录了15个粒子，那么粒子击中计数器的实际数量n 的后验分布是什么？粒子平均发射速率r 的后验分布是什么？

要解决这样一个问题，我们要考虑系统以这些参数为开始，以观测到的数据为结束之间的关系链：

1．放射源以平均速率r 发射粒子。

2．在任何给定的1秒内，放射源向计数器发射了n 个粒子。

3．n 个粒子中，只有其中k 个被记录下来。

原子衰变的概率在任何时间都是相同的，所以放射性衰变可以很好建模为一个泊松过程。已知r，则n 的分布是参数为r 的泊松分布。

并且如果我们假设检测到每个粒子的概率是独立的，k 的分布即是参数为n 和f 的二项分布。

给定系统的参数，可以求得数据的分布。所以我们可以用所谓的正向问题 （直接思路）来解决。

现在，我们希望用另一个方法：已知数据，求得参数的分布。这就是所谓的逆向问题 。如果能解决正向问题，你就可以使用贝叶斯方法来解决逆向问题。

14.2 从简单的开始

让我们从问题的一个简单的版本——已知r 值——开始。给定f 的值，所以要做的就是估计n 。

我定义了一个名为`Detector` 的Suite对象，对检测器建模并估算n 。

```
class Detector(thinkbayes.Suite):

    def __init__(self, r, f, high=500, step=1):
        pmf = thinkbayes.MakePoissonPmf(r, high, step=step)
        thinkbayes.Suite.__init__(self, pmf, name=r)
        self.r = r
        self.f = f
```

如果平均发射速率为每秒r 个粒子，则n 的分布是参数为r 的泊松分布。`high` 和`step` 定义为n 的上界和假设值的步长大小。

现在我们需要一个似然函数：

```
# class Detector

    def Likelihood(self, data, hypo):
        k = data
        n = hypo
        p = self.f

        return thinkbayes.EvalBinomialPmf(k, n, p)
```

`data` 是检测到的粒子数量，`hypo` 是发射出的粒子的假设数量。

如果实际上有n 个粒子，并且检测到它们中的任何一个的概率为f ，则检测到k 个粒子的概率由二项分布给出。

这就是检测器对象了。我们可以试着求出r 值的范围：

```
    f = 0.1
    k = 15
```

```
    for r in [100, 250, 400]:
        suite = Detector(r, f, step=1)
        suite.Update(k)
        print suite.MaximumLikelihood()
```

图14-1显示了*n* 对于几个给定*r* 值的后验分布。

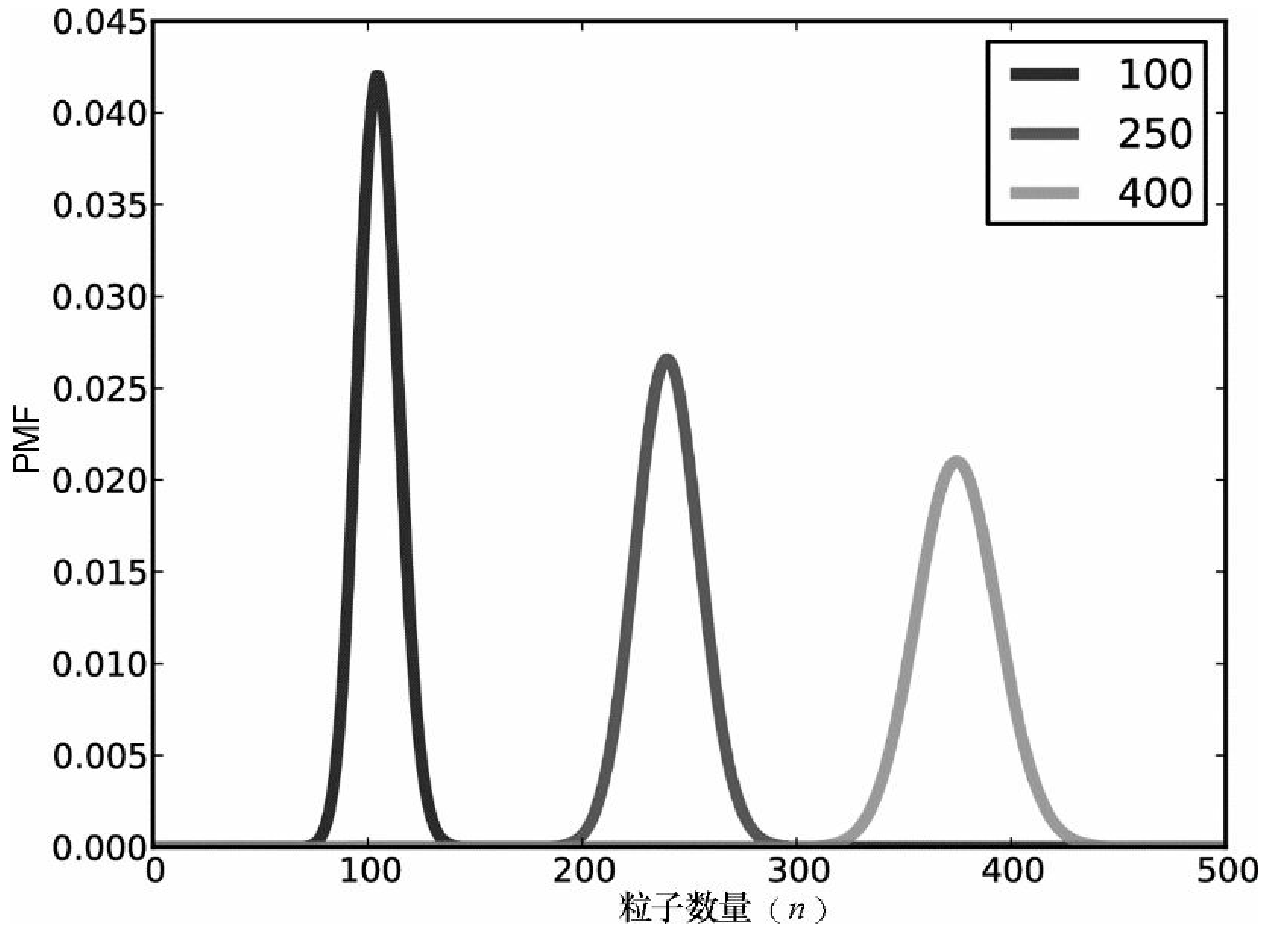


图14-1　3个不同*r* 值下*n* 的后验分布。

14.3　分层模型

在上一节中，我们假设*r* 为已知的。现在，让我们放松这一假设。我定义了另一个Suite对象，称为`Emitter`，即对发射器建模并估计*r* 的范围：

```
class Emitter(thinkbayes.Suite):

    def __init__(self, rs, f=0.1):
        detectors = [Detector(r, f) for r in rs]
        thinkbayes.Suite.__init__(self, detectors)
```

rs 是*r* 假设值的序列。detectors 是检测器对象的序列，每一个对应于一个*r* 值。对象中就是检测器的每个值，所以Emitter是一个元 **Suite** 对象；也就是说，它是以其他Suite对象为值的Suite对象。

要更新Emitter，我们必须计算每个*r* 的假设值下的数据的似然度。但*r* 的各值是由一个包含了*n* 值范围的检测器对象表示的。

要计算对于给定的检测器下数据的似然度，我们通过循环*n* 的所有值，然后累加*k* 的总概率。SuiteLikelihood 实现这个功能：

```
# class Detector

    def SuiteLikelihood(self, data):
        total = 0
        for hypo, prob in self.Items():
            like = self.Likelihood(data, hypo)
            total += prob * like
        return total
```

现在我们可以写出发射器的似然函数：

```
# class Detector

    def Likelihood(self, data, hypo):
        detector = hypo
        like = detector.SuiteLikelihood(data)
        return like
```

每一个hypo 是一个检测器，所以我们可以调用SuiteLikelihood 得到假设下数据的似然度。

更新了发射器后，我们也必须更新每个探测器。

```
# class Detector

    def Update(self, data):
        thinkbayes.Suite.Update(self, data)

        for detector in self.Values():
            detector.Update()
```

像这样有多层Suite对象的模型被称为分层模型 。

14.4 一个小优化

你也许对SuiteLikelihood 有印象；我们在第110页“来一个公平的对比”中见过它。当时我指出我们并不真的需要它，因为由SuiteLikelihood 计算的总概率是由Update 计算并返回的归一化常数。

所以，不用先更新发射器，再更新探测器，我们其实可以同时完成这两步，使用从Detector.Update 得到的结果作为发射器的似然度。

下面是Emitter.Likelihood 的精简版：

```
# class Emitter

    def Likelihood(self, data, hypo):
        return hypo.Update(data)
```

以这个版本的Likelihood ，我们就可以使用Update 的默认版本。因此代码行更少，而且因为它不用计算归一化常量两次，所以运行得更快。

14.5 抽取后验

更新了发射器后，我们可以通过循环探测器和其概率得到r 的后验分布：

```
# class Emitter
```

```
    def DistOfR(self):
        items = [(detector.r, prob) for detector, prob in self.Items()]
        return thinkbayes.MakePmfFromItems(items)
```

`items` 为r 的值及其概率的列表。其结果是r 的Pmf。

为了得到n 的后验分布，我们必须计算出探测器的混合分布。我们可以使用`thinkbayes.MakeMixture`，它接收映射每个分布和其概率的元Pmf。这实际上也就是发射器：

```
# class Emitter

    def DistOfN(self):
        return thinkbayes.MakeMixture(self)
```

图14-2显示了结果。毫不奇怪，n 最可能的值是150。已知f 和n，则预计数为$k = fn$，所以给定f 和k，n 的期望值为k/f。也就是150。

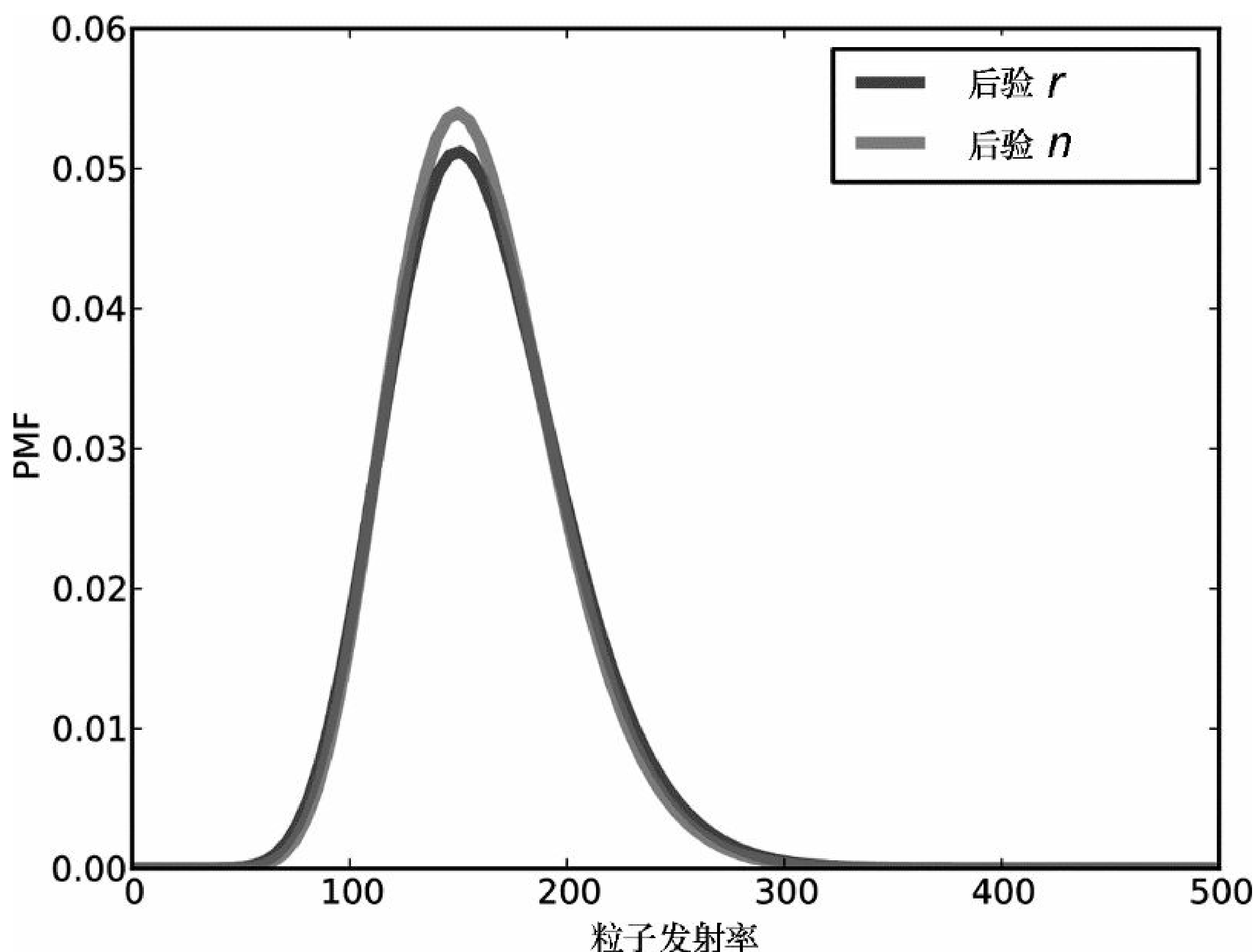


图14-2　n 和r 的后验分布

如果150个粒子在1秒内被发射，r 的最可能的值是每秒150个粒子。因此r 的后验分布也集中在150附近。

r 的后验分布和n 是相似的；唯一的区别是，我们对于n 稍不确定。一般来说，我们对于较长时间范围的发射率r 更确定，但对于任意1秒内的粒子发射数量n 却不是那么确定。

你可以从 *http://thinkbayes.com/jaynes.py* 下载本章代码。更多信息，请参见前言的“代码指南”。

14.6　讨论

盖革计数器问题表明因果关系和分层模型之间的联系。在该示例中，发射率r 对粒子数n 有因果效应，而n 对粒子计数量k 有因果效应。

分层模型反映了系统的结构，在顶部产生影响，于底部得到效果。

1．在顶层，我们以r 的一系列假设值开始。

2．对每一个r 的值，我们有一个n 的值的范围，这取决于n 的先验分布。

3．当更新模型时，我们自下而上。对于每个r 值计算n 的后验分布，然后计算r 的后验分布。所以因果信息沿着层次结构由上至下，而推断过程自底向上。

14.7 练习

练习14-1。

这项工作也是受到杰恩斯《概率论》书中一个例子的启发。

假设你买了一个预期能降低家里附近蚊虫数量的捕蚊器。每周你都清空这个陷阱，计算抓获的蚊虫数量。第一个星期后捉到30只蚊子。第二个星期后捉到20只蚊子。那么请估计你的院子里蚊子数量的百分比变化。

要回答这个问题，必须做出一些建模的决定。这里有一些建议如下：

- 假设每个星期有大量蚊子N 在你家附近的湿地生存。
- 一周之内，N 中一部分f_1 进入你的院子里，f_1 中一部分f_2 落入陷阱。
- 在你的方法中，要考虑到“N 逐周可能的变化量”的先验观点，可以通过在分层模型中增加一个层次来对N 变化的百分比建模。

第15章　处理多维问题

15.1　脐部细菌

肚脐生物多样性2.0（BBB2）项目是一个全国性的民间科学项目，旨在识别可以在人类肚脐上找到的细菌种类（ *http://bbdata.yourwildlife.org* ）。该项目似乎异想天开，但它是人们越来越关注人体微生物的趋势的一部分，人体微生物就是那些生活在人体皮肤与身体各部分的微生物的集合。

在试验性研究中，BBB2研究人员收集了60名志愿者脐部的药签，用复用焦磷酸测序法提取并进行16S rDNA片段的测序，然后确定其物种基因片段的来源。每一个识别出的片段被称为“标记样本” ① 。

我们可以利用这些数据来回答几个相关问题：

- 基于观察到的物种的数量，我们能否估算在环境中物种的总数？
- 我们能否估算每一个物种的种群比例，即每一个物种占总体的分数？
- 如果我们计划收集额外的样本，能否预测有多少新物种可能会被发现？
- 要使观察到的物种的比例增加到一个给定的阈值，需要多少额外的“标记样本”片段？

这些问题构成了所谓的未知物种问题 。

15.2　狮子，老虎和熊

我将从这个问题的一个简化版本开始。在这个版本中，我们已知物种的情况，姑且称之为狮子、老虎和熊。假设我们参观野生动物保护区，看到了3只狮子、2只老虎和1头熊。

如果我们在保护区观察到任何动物物种的机会均等，则每个物种的数量由多项分布决定。假设狮子、老虎和熊的种群比率是`p_lion`

、p_tiger 和p_bear ，看到3只狮子，2只老虎和1头熊的可能性就是

```
p_lion ** 3 * p_tiger ** 2 * p_bear ** 1
```

一种诱人但不正确的方法是用beta分布（见32页的“Beta分布”）来分别描述每个物种的种群比例。例如，我们看到3只狮子和3只“非狮子”；如果我们将其视作3个“正面”和3个“反面”的话，那么p_lion 的后验分布就是：

```
    beta= thinkbayes.Beta ()
    beta.Update ((3,3))
    print beta.MaximumLikelihood ()
```

p_lion 的最大似然估计就是观察到的比例50%。同样，p_tiger 和p_bear 的最大极大似然估计为33%和17%。

但这里有两个问题：

1. 我们已经隐含地为每个物种使用了一个均匀的从0到1的先验，但是因为我们知道有3个品种，所以其实这个先验是不正确的。正确的先验应该是平均值为1/3，并且在（其他的）物种具有100%的种群比例时似然度应该为零。

2. 每个物种的分布不是独立的，因为种群比例总和为1。为了体现这种依赖，我们需要3个物种种群比例的联合分布。

可以用一个狄利克雷解决这两个问题（见*http://en.wikipedia.org/wiki/Dirichlet_distribution* ）。就如我们以beta分布来描述不均匀硬币的分布一样，我们可以使用狄利克雷分布来描述p_lion ，p_tiger 和p_bear 的联合分布。

狄利克雷分布是beta分布的多维通用版本。与正面、反面这种双值结果不同，狄利克雷分布能处理任何数量的结果：在这个例子中，是3个物种。

如果有n 个结果，狄利克雷分布是由n 个参数描述的，记为α_1 到α_n 。

thinkbayes.py 中，有一个定义了狄氏的类如下：

```
class Dirichlet(object):

     def __init__(self, n):
         self.n = n
         self.params = numpy.ones(n, dtype=numpy.int)
```

n 为维数；最初的参数都是1，我用numpy 数组存储参数，这样我可以利用数组操作的优势。

给定一个狄利克雷分布，每个种群比例的边缘分布是一个beta分布，我们可以计算如下：

```
    def MarginalBeta(self, i):
        alpha0 = self.params.sum()
        alpha = self.params[i]
        return Beta(alpha, alpha0-alpha)
```

i 是我们想要的边缘分布的指数。alpha0 是参数的总和；alpha 是对于给定物种的参数。

在该示例中，每个物种的前验边缘分布为Beta(1，2)。我们可以计算平均前验如下：

```
    dirichlet = thinkbayes.Dirichlet(3)
    for i in range(3):
        beta = dirichlet.MarginalBeta(i)
        print beta.Mean()
```

正如预期的那样，每个物种种群比例的前验均值是1/3。

要更新狄利克雷分布，我们把这一观察结果添加到参数：

```
    def Update(self, data):
        m = len(data)
        self.params[:m] += data
```

这里data是和params顺序一致的一个计数序列，所以在这个例子中，它应该是狮子、老虎和熊的数量。

data可以比params短；在这种情况下，意味着有一些物种没有被观察到。

下面是以观察到的数据更新dirichlet并计算后验边缘分布的代码。

```
data = [3, 2, 1]
dirichlet.Update(data)

for i in range(3):
    beta = dirichlet.MarginalBeta(i)
    pmf = beta.MakePmf()
    print i, pmf.Mean()
```

图15-1显示了结果。平均种群比例的后验是44%、33%和22%。

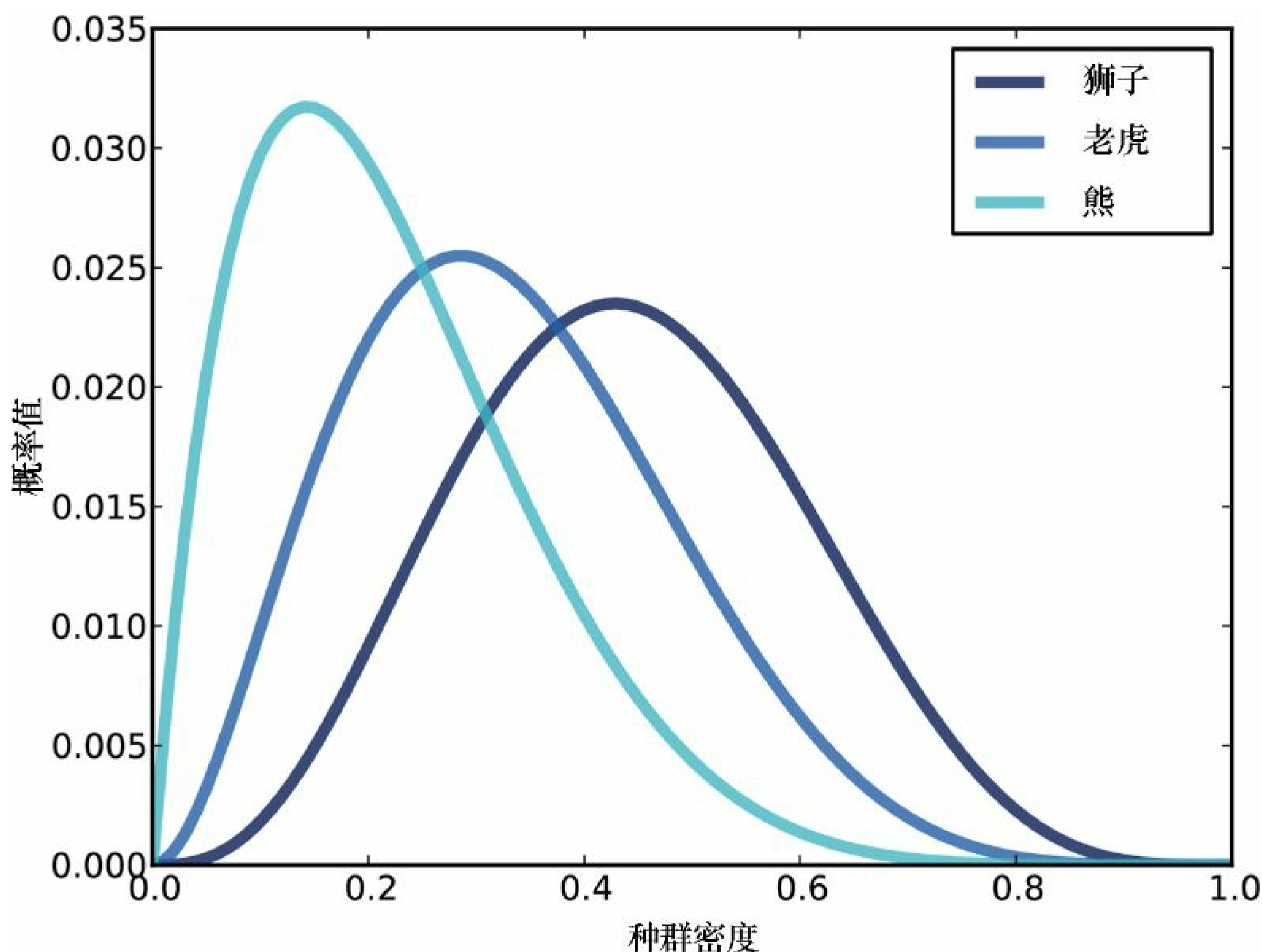


图15-1　三个物种种群比例的分布

15.3　分层版本

我们已经解决了这个问题的一个简化版本：如果我们知道有多少物种，我们可以估算每一个的种群比例。

现在让我们回到原来的问题，估计物种的总数。要解决这个问题，我会定义一个元Suite对象，它是一个包含其他Suite对象作为假设的Suite对象。在这个例子里，最上层的Suite对象包含物种数量的假设，底层包含种群比例的假设。

类定义如下：

```
class Species(thinkbayes.Suite):
```

```
    def __init__(self, ns):
        hypos = [thinkbayes.Dirichlet(n) for n in ns]
        thinkbayes.Suite.__init__(self, hypos)
```

`__init__` 取为n 的可能值的列表，并创建一个狄利克雷对象的列表。

下面是创建顶层Suite对象的代码：

```
ns = range(3, 30)
suite = Species(ns)
```

ns 是n 的可能值的列表。由于已经看到3个物种，所以必须为至少3个。我选择了一个似乎合理的上限，稍后会检查得知超过这个界限的概率很低。并且至少在最初阶段，我们假定任何在此范围内的值都是等可能的。

要更新一个分层模型，你必须更新所有层次。通常必须先更新底层再向上更新，但在本例中，我们可以先更新顶层：

```
#class Species

   def Update(self, data):
       thinkbayes.Suite.Update(self, data)
       for hypo in self.Values():
           hypo.Update(data)
```

`Species.Update` 调用父类中的`Update`，然后遍历子假设并更新它们。

现在，我们需要一个似然函数：

```
# class Species

     def Likelihood(self, data, hypo):
         dirichlet = hypo
         like = 0
         for i in range(1000):
```

```
            like += dirichlet.Likelihood(data)

        return like
```

data 是观察到的计数序列；hypo 是一个狄利克雷对象。Species.Likelihood 调用Dirichlet.Likelihood 共1000次然后返回总和。

为什么调用1000次？因为Dirichlet.Likelihood 实际上并不计算数据在整个狄利克雷分布上的似然度。相反，它从假设的分布中取得一个样本然后计算数据在这个种群比例样本集下的似然度。

下面是实例：

```
# class Dirichlet

    def Likelihood(self, data):
        m = len(data)
        if self.n < m:
            return 0
        x = data
        p = self.Random()
        q = p[:m]**x
        return q.prod()
```

data 的长度是观察到的物种的数量。如果看到的物种比我们预计存在的多，似然度就是0。

否则，我们随机选择一组种群比例p ，再计算多项式Pmf，也就是：

$$c_x p_1^{x_1} \cdots p_n^{x_n}$$

p_i 为第i 个物种的种群比例，x_i 是所观察到的数量。第一项c_x 是多项式系数；我将其放置在在计算之外，因为它仅依赖于数据的乘法因子而不是假设，所以它被归一化了（见*http://en.wikipedia.org/wiki/Multinomial_distribution* ）。

m 是观察到的物种的数量。我们只需要p 的前m 个元素。至于其他

的部分：x_i 为0，所以$p_i^{x_i}$ 为1，我们可以在结果中单列。

15.4 随机抽样

有两种方法可以从狄利克雷分布产生随机样本。一个是使用边缘beta分布，但在这种情况下，你必须一次选取一个值再扩展到余下的值，使得它们累加和为1（参见*http://en.wikipedia.org/wiki/Dirichlet_distribution#Random_number_generati*）。

另一个没那么明显但速度更快的方法是从`n` 个伽玛（gamma）分布中选取值，然后通过除以总和来归一化。下面是代码：

```
# class Dirichlet

    def Random(self):
        p = numpy.random.gamma(self.params)
        return p / p.sum()
```

现在，我们准备好查看结果了。下面是提取`n` 的后验分布的代码：

```
    def DistOfN(self):
        pmf = thinkbayes.Pmf()
        for hypo, prob in self.Items():
            pmf.Set(hypo.n, prob)
        return pmf
```

`DistOfN` 通过在顶层假设中迭代，并且累加每个`n` 的概率。

图15-2显示了结果。最可能的值是4，3到7之间的值也很有可能；之后的概率就迅速下降了。有29个物种的概率低到足以忽略不计；如果我们选择了一个更高的上界，也会得到相当一致的结果。

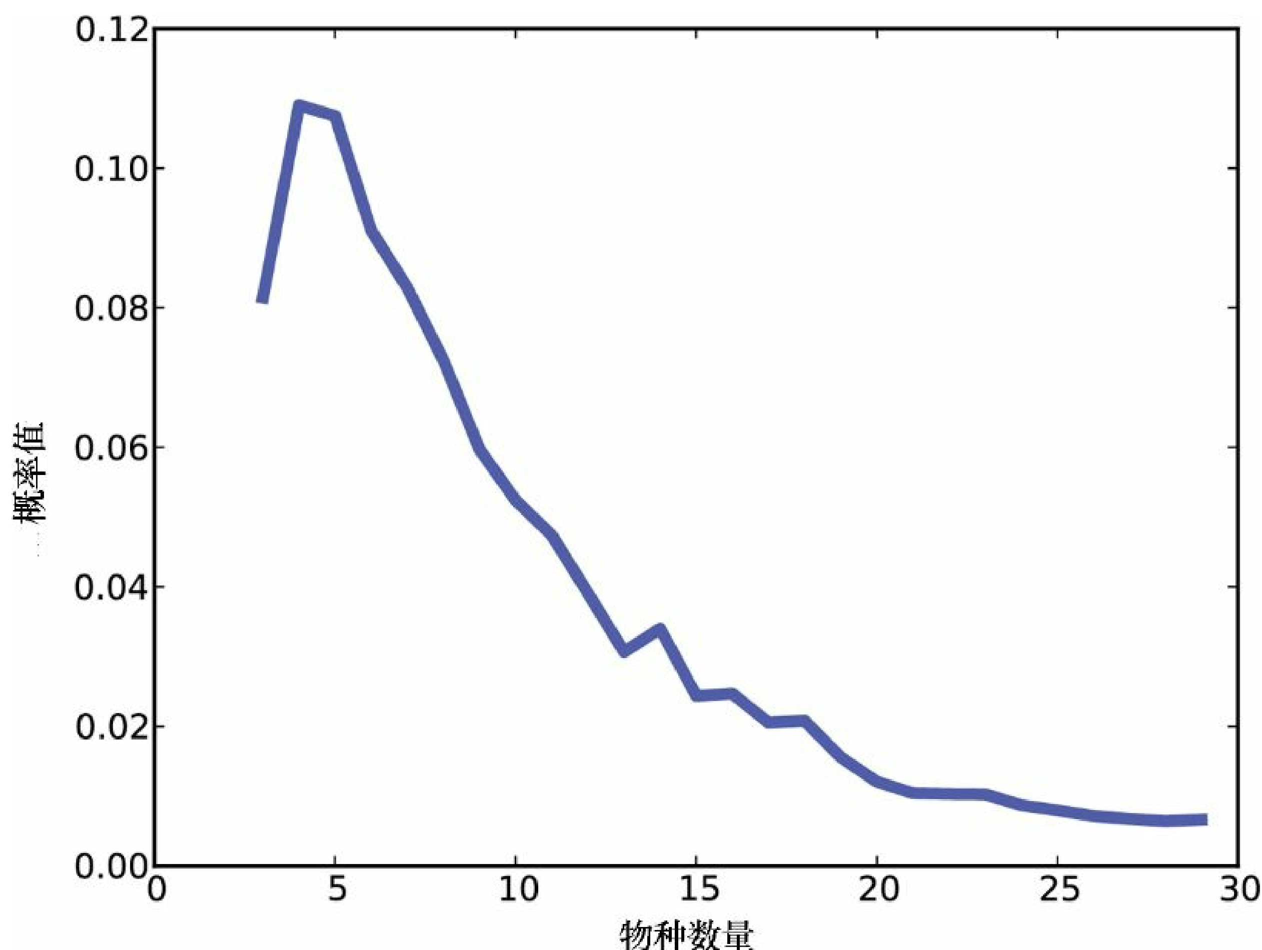


图15-2　n的后验分布

请记住，这个结果基于n 的先验是均匀分布的。如果我们知道物种在环境中数量的背景信息，我们可以选择一个不同的先验 。

15.5　优化

必须承认，我对于这个例子中的解法很自豪。未知物种问题并不简单，而这个解法简单明了，只用了少的难以置信代码（约50行）。

它唯一的问题是计算慢。对于只有3个观察物种的例子它足够好，但对于肚脐问题的数据就不够好了——这类数据的采样中有超过100个物种。

接下来的几节介绍了一系列扩展这个解法的优化方法。在深入到细节里面前，路线图如下。

- 第一步是要认识到，如果我们以同样的数据更新狄利克雷分布，对于所有的值，前m 个的参数都是相同的。唯一的区别是假设的未知物种数量。因此，我们并不真正需要n 个狄利克雷对象；我们可以在层次结构上存储最上层的参数。Species2 实现了这个优化。
- Species2 也对所有假设使用了相同的一组随机值。这节省了生成随机值的时间，但它还有更重要的第二个好处：通过从样品空间给所有的假设赋予相同的选取值，使得在假设之间的比较更加公平，所以它只需较少的迭代次数就能收敛。
- 即使有了这些改动，还有一个重大的性能问题。随着观察到的物种的增加，随机种群比例的数组也变大了，而选取到一个近似正确的值的几率就变小了。所以绝大多数迭代次数得到的似然度很小，以致对总和产生的贡献不多，也就没有在假设之间产生区别。

  解决方法是每次只更新一个物种，Species4 是使用狄利克雷对象来表示子假设这一策略的一个简单实现。

- 最后，在模型顶层Species5 结合子假设，并使用numpy 数组运算以加快速度。

如果你对细节不感兴趣，可以跳到156页的“肚脐数据”，在那儿查看来自肚脐数据的结果。

15.6 堆叠的层次结构

所有底层的狄利克雷分布以相同的数据更新，所以对它们来说，前m 个参数相同。我们可以通过将参数合并入顶层Suite对象来消除这一重复过程。Species2 实现了这一优化：

```
class Species2(object):

    def __init__(self, ns):
        self.ns = ns
        self.probs = numpy.ones(len(ns), dtype=numpy.double)
        self.params = numpy.ones(self.high, dtype=numpy.int)
```

ns 是n 假设值的一个列表；probs 是相应概率的列表。而params 是狄氏参数的顺序，初始所有的参数都为1。

Species2.Update 更新这一分层（模型）的两个层次。第一层次是n 的每个可能值的概率，下一层次是狄利克雷参数：

```
# class Species2

    def Update(self, data):
        like = numpy.zeros(len(self.ns), dtype=numpy.double)
        for i in range(1000):
            like += self.SampleLikelihood(data)

        self.probs *= like
        self.probs /= self.probs.sum()

        m = len(data)
        self.params[:m] += data
```

SampleLikelihood 返回似然度的一个数组，每一个似然度值对应于n 的每个可能值。Like 累加1000个样本总的似然度。self.probs 乘以总似然度，然后归一化。最后两行更新参数，和Dirichlet.Update 一样。

现在，让我们来看看SampleLikelihood ，这里有两个可优化的地方。

- 当物种的假想数n 超过所观察到的数目m ，我们只需要多项式PMF 的前m 个项；其余均为1。
- 如果物种的数量很大，该数据的似然度用浮点数来表示可能太小（请参阅99页的“数据下溢”）。因此，计算对数似然度要更安全。

同样，该多项式的PMF是

$$c_x p_1^{x_1} \cdots p_n^{x_n}$$

所以对数似然度是

$$\log c_x + x_1 \log p_1 + \cdots + x_n \log p_n$$

它更快速且容易计算。同样，c_x 对所有假设是相同的，所以我们可以放在一边先不管它。下面是代码：

```
# class Species2

    def SampleLikelihood(self, data):
        gammas = numpy.random.gamma(self.params)
        m = len(data)
        row = gammas[:m]
        col = numpy.cumsum(gammas)

       log_likes = []
        for n in self.ns:
            ps = row / col[n-1]
            terms = data * numpy.log(ps)
            log_like = terms.sum()
            log_likes.append(log_like)

        log_likes -= numpy.max(log_likes)
        likes = numpy.exp(log_likes)

        coefs = [thinkbayes.BinomialCoef(n, m) for n in self.ns]
        likes *= coefs

        return likes
```

gammas 是一个伽玛分布值构成的数组，它的长度是n 假设值的最大值。row 是gammas 的前m 个元素；因为这些参数仅依赖于数据本身，所以这些就是我们需要的全部参数了。

对于n 的每个值，我们需要将row 除以gamma 中前n 项值的总和，cumsum 计算这些累积值，并将它们存储在col 中。

loop循环迭代n 的这些值，再累加到一个对数似然值的列表。

在循环内部，ps 包含概率的行数，以对应的累计总和进行归一。terms 包含和的各项，$x_i \log p_i$，log_like 包含它们的和。

循环结束后，我们要将对数似然度转化为线性似然度，但最好是先进行转置使得最大的对数似然度为0；这样的话，线性似然度就不会显得太小（请参阅“数据下溢”99页）。

最后，在返回似然度前，我们必须应用一个修正系数（因子），它是我们可以观察到m 个物种的可能方法的数量，同时假设物种的总数

为n 。BinomialCoefficient 计算这个“*n* 选*m* ”过程，写为$\binom{n}{m}$ 。

正如常见的那样，优化的版本可读性差，也比原始版本更容易出错。但这就是我从简单版本开始的一个原因，我们可以用它进行回归测试。我绘制了从两个版本得到的结果值，可以确认它们是大致相等的，并且随着迭代次数增加它们都是收敛的。

15.7 另一个问题

我们还可以做得更多来优化这一代码，但有另外一个需要首先处理的问题。随着观察到的物种数目的增加，这个版本的解法变得让人心烦，它需要更多的迭代才能收敛到一个好的结果。

但问题是，如果我们从狄利克雷分布中选择的种群比例ps 并不是近似正确的，所观察到的数据的似然度就会接近零，而对于n 的所有值几乎就是错的。那么大多数迭代就不能为总体可能性提供有用的贡献。随着所观察到的物种的数量m 变大，以确切可能性选取ps 的概率就会变小，它真是相当小。

幸运的是，有一个解决办法。如果你观察一组数据，你可以就整个数据集更新先验分布，或者把它分解成一系列所述数据的子集再逐一更新，并且这两种方式更新的结果都是相同的。

在这个例子中，关键是每次更新是针对一个物种。这样，当我们生成一组随机的ps 时，只有其中一个会影响到计算得到的似然度，因此选择一个正确对象的概率要好得多。

下面是每次更新一个物种的新版本：

```
class Species4(Species):

    def Update(self, data):
        m = len(data)

        for i in range(m):
            one = numpy.zeros(i+1)
            one[i] = data[i]
            Species.Update(self, one)
```

该版本从Species 继承__init__ ，所以它以一个狄利克雷对象的列表来表示假设（和Species2 中不同）。

Update 遍历观察到的物种，创建一个数组one ，它包含某一个物种的计数，都先预置为零。然后调用父类的Update ，计算似然度并更新子假设。

因此，在这个例子里，我们做三次更新。第一个有点像“看到了三只狮子”。第二个是“看到了两只老虎，没有看到更多的狮子”。第三个是“看到一头熊，没有看到更多狮子和老虎”。

下面是似然度的新版本：

```
# class Species4

    def Likelihood(self, data, hypo):
        dirichlet = hypo
        like = 0
        for i in range(self.iterations):
            like += dirichlet.Likelihood(data)

        # correct for the number of unseen species the new one
        # could have been
        m = len(data)
        num_unseen = dirichlet.n - m + 1
        like *= num_unseen

        return like
```

这和Species.Likelihood 几乎相同，所不同的是因子num_unseen 。这种校正是必要的，因为每当我们首次看到某一个物种，我们都要考虑到有其他的一些我们本应看到的未见物种。对于较大的n 值就会有更多我们本应看到的未见物种，这一因素增加了数据的似然度。

必须承认这是一个我最开始并没有搞清楚的微妙之处，但搞清楚后，紧接着我就能够通过比较以前的版本来验证它（该版本）了。

15.8 还有工作要做

每次执行一个物种的更新解决了一个问题，但它也带来了另一个问题。每次更新需要的时间正比于km，其中k是假设的数目，m是观察到的物种的数量。因此，如果我们做m次更新，总运行时间正比于km^2。

但是，我们可以利用151页“堆叠的层次结构”中同样的诀窍加快速度：我们将摆脱狄利克雷对象，将分层结构中的两个层次折叠到一个单一的对象。所以下面是Species 的另一个版本：

```
class Species5(Species2):

    def Update(self, data):
        m = len(data)
        for i in range(m):
            self.UpdateOne(i+1, data[i])
            self.params[i] += data[i]
```

该版本从Species2 继承__init__，所以它使用ns 和probs 表示n的分布，而params 表示狄利克雷分布的参数。

Update 类似于我们在上一节中看到的。它遍历观察到的物种再调用UpdateOne：

```
# class Species5

    def UpdateOne(self, i, count):
        likes = numpy.zeros(len(self.ns), dtype=numpy.double)
        for i in range(self.iterations):
            likes += self.SampleLikelihood(i, count)

        unseen_species = [n-i+1 for n in self.ns]
        likes *= unseen_species

        self.probs *= likes
        self.probs /= self.probs.sum()
```

此函数类似于Species2.Update，但有两个变化。

- 接口是不同的。我们得到的不是整个数据集，而是观察到的物种的索引i 和已经看到的物种数量count 。
- 我们要在未见物种的数量上应用一个修正系数，如

Spcies4.Likelihood 所示。这里的不同之处在于我们以数组的乘法即时更新全部似然度。

最后，SampleLikelihood 如下：

```
# class Species5

    def SampleLikelihood(self, i, count):
        gammas = numpy.random.gamma(self.params)

        sums = numpy.cumsum(gammas)[self.ns[0]-1:]

        ps = gammas[i-1] / sums
        log_likes = numpy.log(ps) * count

        log_likes -= numpy.max(log_likes)
        likes = numpy.exp(log_likes)

        return likes
```

这类似于Species2.SampleLikelihood；不同的是，每一次更新只包括单一物种，所以并不需要一个循环。

这个函数的运行时间正比于假设数量k 。它运行m 次，因此更新的运行时间成正比于km 。所以我们得到一个准确结果需要的迭代次数通常也就少了。

15.9　肚脐数据

关于狮子、老虎和熊的问题讨论得已经足够了。现在让我们回到肚脐问题。为了得到数据的含义，考虑B1242课题，其400个标记样本产生了如下计数的61个物种：

```
92, 53, 47, 38, 15, 14, 12, 10, 8, 7, 7, 5, 5,
4, 4, 4, 4, 4, 4, 4, 3, 3, 3, 3, 3, 3, 3, 2, 2, 2, 2,
1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1
```

有少数几个优势物种构成了整体的很大一部分，而很多其他物种只

产生一个单一的样本标记。这些“单身”的数目表明了没观察到的物种可能至少有那么几个。

狮子和老虎的例子中，我们假设在保护区里每种动物被观测到的可能性是相等的。类似地，对于肚脐问题的数据，我们假设每种细菌被标记到的可能也是相等的。

在现实中，数据收集过程中每个步骤都可能引入偏差。有些物种被拭子拾起的可能性大些，或容易产生可供识别的扩增子（生物遗传学名词：聚合酶链式反应获得的双链核苷酸产物）。所以当我们谈到每一个物种的种群比例时，我们应当要考虑到这种误差的来源。

还要承认，我不太严格地使用了术语“物种”。首先，细菌物种没有得到很好的限定。其次，有的样本标记能够识别特定的物种，其他的则只能识别一个属。更准确地，我应该说“操作分类单位”，或简写为OTU（operational taxonomic unit）。

现在，让我们来处理一些肚脐数据。我定义了一个Subject 类表示研究中每个相关课题的信息：

```
class Subject(object):

    def __init__(self, code):
        self.code = code
        self.species = []
```

每个课题都有一个字符编码，比如“B1242”，还有（计数，种名）对的列表，按计数递增的顺序存放。Subject 提供了几个方法，可以很容易地得到这些计数和物种的名称。你可以从 http://thinkbayes.com/species.py 了解细节。更多信息，请参见前言的“代码指南”。

Subject 提供了名为Process 的方法来创建和更新一个Species5 的Suite对象，它代表n 的分布和种群比例。

Suite2 提供了DistOfN 方法，它返回n 的后验分布。

```
# class Suite2
```

```
    def DistN(self):
        items = zip(self.ns, self.probs)
        pmf = thinkbayes.MakePmfFromItems(items)
        return pmf
```

图15-3显示了课题B1242中n 的分布。刚好有61个物种，即没有未见物种的概率几乎为零。物种数量最可能的值是72，90%置信区间处在66到79之间。从横坐标的高值部分来看，有87个物种的可能性很小。

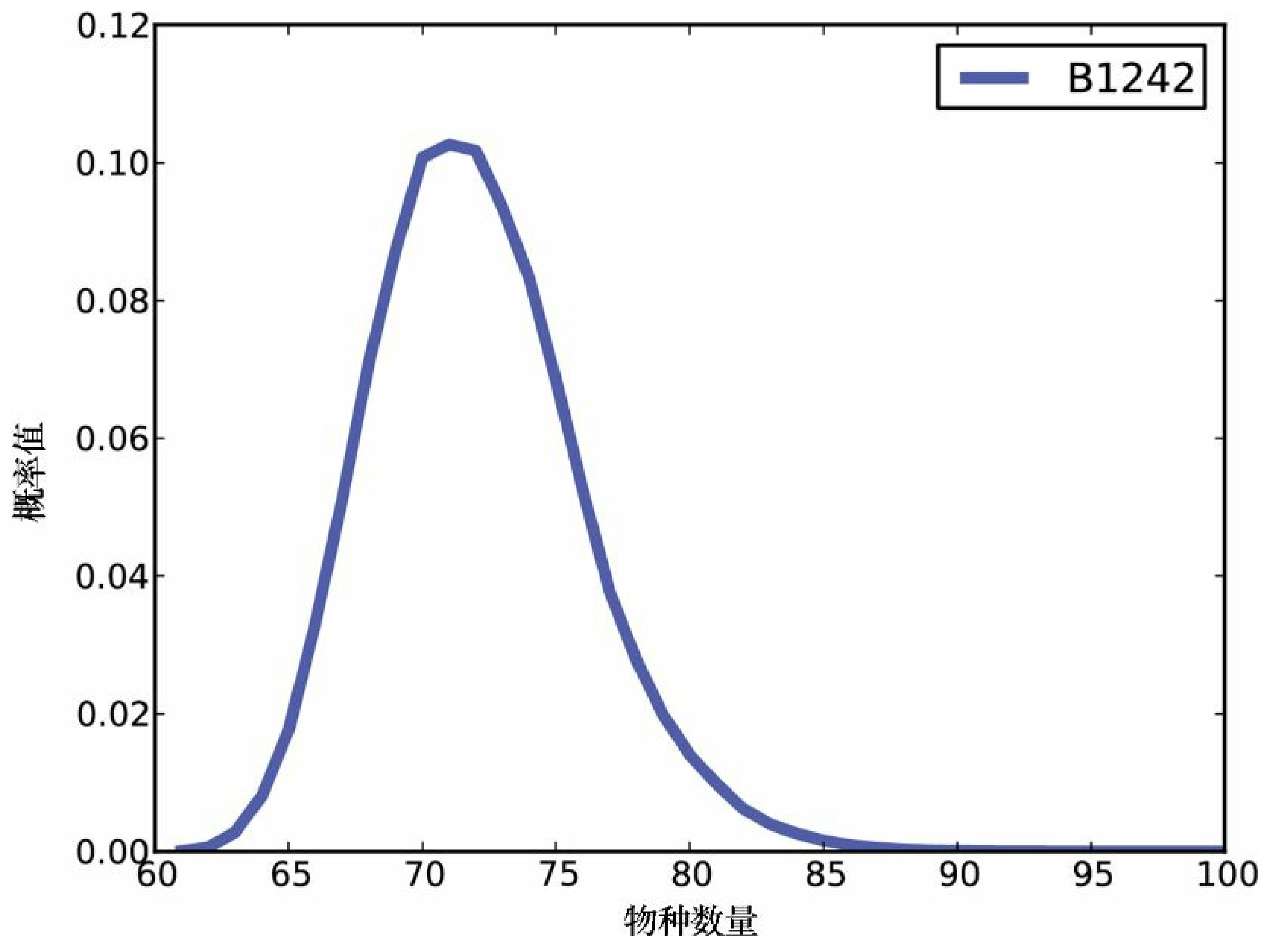


图15-3　课题B1242中n的分布

接下来我们计算每个物种种群比例的后验分布。Species2 提供了DistOfPrevalence：

```
# class Species2
```

```
    def DistOfPrevalence(self, index):
        metapmf = thinkbayes.Pmf()

        for n, prob in zip(self.ns, self.probs):
            beta = self.MarginalBeta(n, index)
            pmf = beta.MakePmf()
            metapmf.Set(pmf, prob)

        mix = thinkbayes.MakeMixture(metapmf)
        return metapmf, mix
```

index 显示了我们想要计算的物种。对于每个n，种群比例的后验分布不同。

循环迭代n 的可能值及其概率。由n 的每一个值得到表示所指物种边缘分布的一个Beta对象。别忘了Beta对象包含有参数alpha 和beta；它们不像Pmf对象那样有值和其概率对，但它们提供MakePmf，生成近似于连续Beta分布的离散值。

metapmf 是一个在已知n 的条件下，包含种群比例分布的元Pmf。MakeMixture 将元Pmf结合到mix，mix 结合条件分布到一个单一的种群比例分布。

图15-4显示了前五个最大物种的结果。这些最常见的物种占到了400个标记样本中的23%，但由于肯定存在未见物种，因此其种群比例最有可能的估计为20%，90%的置信区间是17%～23%。

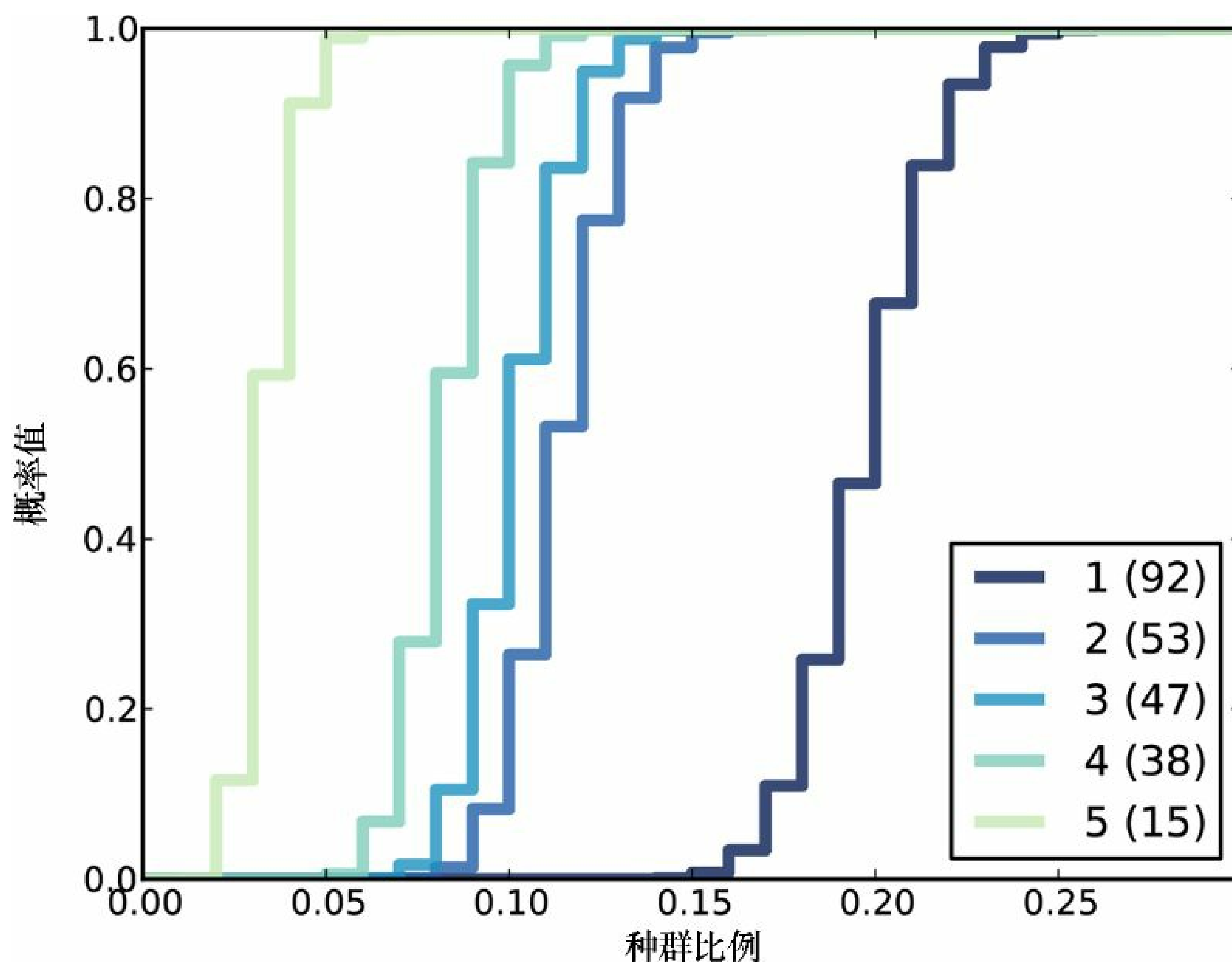


图15-4　B1242种群比例的分布

15.10　预测分布

在4个相关问题中，我介绍了隐藏物种的问题。通过计算n 和每个物种种群比例的后验分布，我们已经回答了前两个。

另外两个问题是：

- 如果计划收集更多的样本，我们能不能预测可能发现多少个新物种？
- 需要增加多少额外标记样本，才能提高观察到物种的比例到一个给定的阈值？

要回答类似这样的预测问题，我们可以使用后验分布来模拟可能的

未来事件，并计算可能看到的物种数量预测的分布，以及总数占比。

这些模拟过程的核心是：

1．从后验分布选取n 值；

2．为每一个物种选取其种群比例，包括可能的未见物种，使用狄利克雷分布；

3．生成未来观测值的随机序列；

4．计算新物种的数量，num_new ，作为额外样本数k 的函数；

5．重复前面的步骤，累加num_new 和k 的联合分布。

下面的代码RunSimulation 运行了一个单次模拟：

```
# class Subject

    def RunSimulation(self, num_reads):
        m, seen = self.GetSeenSpecies()
        n, observations = self.GenerateObservations(num_reads)

        curve = []
        for k, obs in enumerate(observations):
            seen.add(obs)

            num_new = len(seen) - m
            curve.append((k+1, num_new))

        return curve
```

num_reads 是要模拟的额外的样本数。m 是可见物种的数目，而seen 是每个物种唯一名称的字符串集合。n 是一个从后验分布选取的随机值，observations 是物种名字的一个随机序列。

每经过一次循环，我们在seen 中加入得到的新的观察结果，并记录样本数和目前新物种的数量。

RunSimulation 的结果是一个稀疏曲线，表示为样本数和新物种

数目对的一个列表。

检查结果前，我们来看看GetSeenSpecies 和 GenerateObservations 。

```
#class Subject

    def GetSeenSpecies(self):
        names = self.GetNames()
        m = len(names)
        seen = set(SpeciesGenerator(names, m))
        return m, seen
```

GetNames 在数据文件中的返回物种名称的列表，但对于许多课题，这些名称不是唯一的，所以我通过SpeciesGenerator 用序列号来扩展每个名称：

```
def SpeciesGenerator(names, num):
    i = 0
    for name in names:
        yield '%s-%d' % (name, i)
        i += 1

    while i < num:
        yield 'unseen-%d' % i
        i += 1
```

已知一个命名Corynebacterium，SpeciesGenerator 将产生"Corynebacterium-1"这样的名字。当名称列表耗尽时，将产生如unseen-62 这样的命名。

GenerateObservations 如下：

```
# class Subject

    def GenerateObservations(self, num_reads):
        n, prevalences = self.suite.SamplePosterior()

        names = self.GetNames()
        name_iter = SpeciesGenerator(names, n)
```

```
        d = dict(zip(name_iter, prevalences))
        cdf = thinkbayes.MakeCdfFromDict(d)
        observations = cdf.Sample(num_reads)
        return n, observations
```

同样的，num_reads 是要生成的额外样本数。n 和prevalences 是后验分布的样本。

cdf 是一个映射物种名到累积概率的Cdf对象，包括未见物种，采用Cdf使产生物种名称的随机序列过程变得高效。

最后，Species2.SamplePosterior 如下：

```
    def SamplePosterior(self):
        pmf = self.DistOfN()
        n = pmf.Random()
        prevalences = self.SamplePrevalence(n)
        return n, prevalances
```

SamplePrevalences 生成种群比例在条件为n 时的样本：

```
# class Species2

    def SamplePrevalences(self, n):
        params = self.params[:n]
        gammas = numpy.random.gamma(params)
        gammas /= gammas.sum()
        return gammas
```

我们会看到这个算法从狄利克雷分布产生随机值（如第149页“随机抽样”所述的）。

图15-5显示了B1242课题的100个模拟的稀疏曲线。该曲线显得“抖动”是因为我为每条曲线加了一个随机偏移量，使得它们不会重叠在一起。通过观察，我们可以估算出400个额外的标记样本后，我们很可能会发现2～6个新的物种。

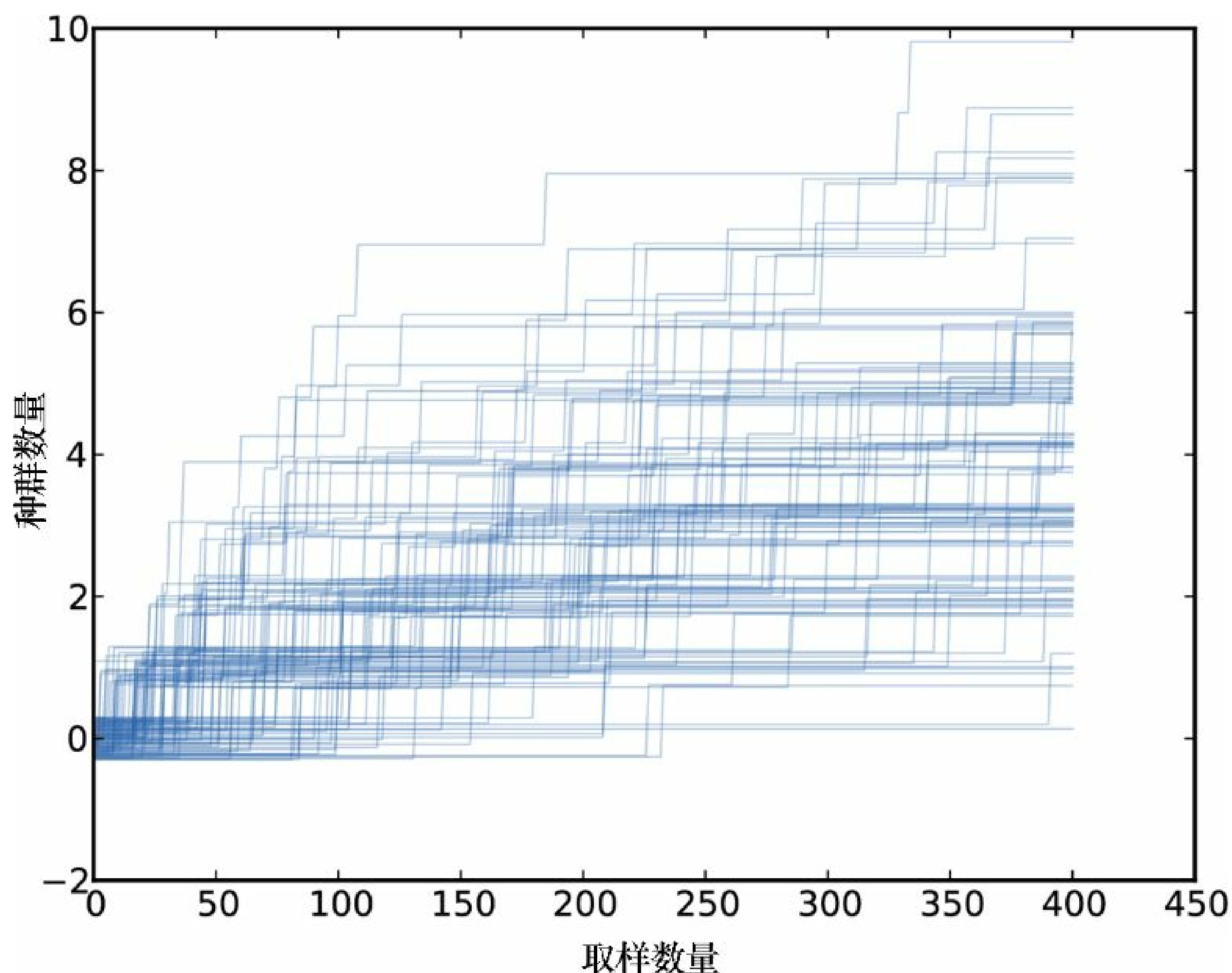


图15-5　B1242课题的模拟稀疏曲线

15.11　联合后验

我们可以利用这些模拟来估计num_new 和k 的联合分布，由此，我们还可以得到num_new 在k 为任意值条件下的分布。

```
def MakeJointPredictive(curves):
    joint = thinkbayes.Joint()
    for curve in curves:
        for k, num_new in curve:
            joint.Incr((k, num_new))
    joint.Normalize()
    return joint
```

MakeJointPredictive 创建一个Joint对象，它是一个以元组为值的Pmf对象。

curves 是一个RunSimulation 创建的稀疏曲线的列表。每条曲线包含k 和num_new 对的列表。

由此产生的联合分布是从每个数值到其发生概率的一个映射关系。给定联合分布，我们可以通过Joint.Conditional 得到num_new 以k 为条件下的分布（见“条件分布”第90页）。

Subject.MakeConditionals 接收一个ks 的列表，并计算num_new 对每个k 的条件分布。其结果是一个Cdf对象构成的列表。

```
def MakeConditionals(curves, ks):
    joint = MakeJointPredictive(curves)

    cdfs = []
    for k in ks:
        pmf = joint.Conditional(1, 0, k)
        pmf.name = 'k=%d' % k
        cdf = pmf.MakeCdf()
        cdfs.append(cdf)

    return cdfs
```

图15-6显示了结果。100个额外的标记样本后，预测的新物种的数量中值为2；90%置信区间为0到5。800个标记样本后，我们有望看到3到12个新的物种。

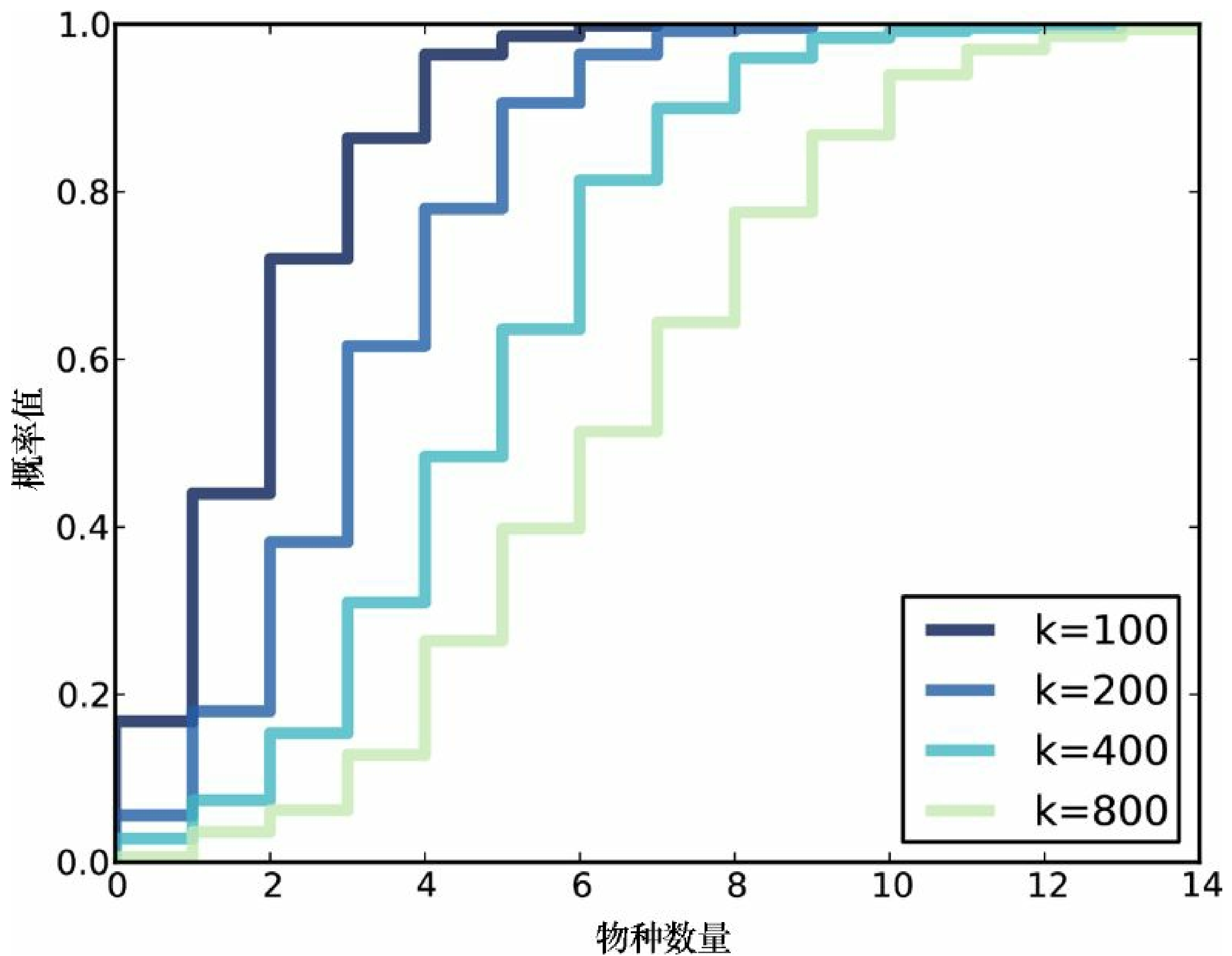


图15-6　在不同数量的额外标记样本条件下，新发现物种数量的分布

15.12　覆盖

我们要回答最后一个问题：“需要增加多少额外的标记样本，才能把观察到物种比例提高到一个给定的阈值”？

要回答这个问题，我们需要RunSimulation的一个能计算已观测物种的分数比例的版本，而不是新物种数量的版本。

```
# class Subject

    def RunSimulation(self, num_reads):
        m, seen = self.GetSeenSpecies()
        n, observations = self.GenerateObservations(num_reads)
```

```
        curve = []
        for k, obs in enumerate(observations):
            seen.add(obs)

            frac_seen = len(seen) / float(n)
            curve.append((k+1, frac_seen))

        return curve
```

接下来，循环每一条曲线并创建一个字典d，映射额外样本数量k到一个fracs 列表，也就是在取得k 个样本后得到的覆盖率值的列表。

```
    def MakeFracCdfs(self, curves):
        d = {}
        for curve in curves:
            for k, frac in curve:
                d.setdefault(k, []).append(frac)

        cdfs = {}
        for k, fracs in d.iteritems():
            cdf = thinkbayes.MakeCdfFromList(fracs)
            cdfs[k] = cdf

        return cdfs
```

这时，我们对k 的每一个值创建了fracs 的Cdf对象，这个Cdf表示了k 个样本后覆盖率的分布。

CDF告诉了你落入给定阈值内的概率，而互补CDF告诉了你超过阈值范围的概率，图15-7显示了不同k 值范围下的互补CDF。

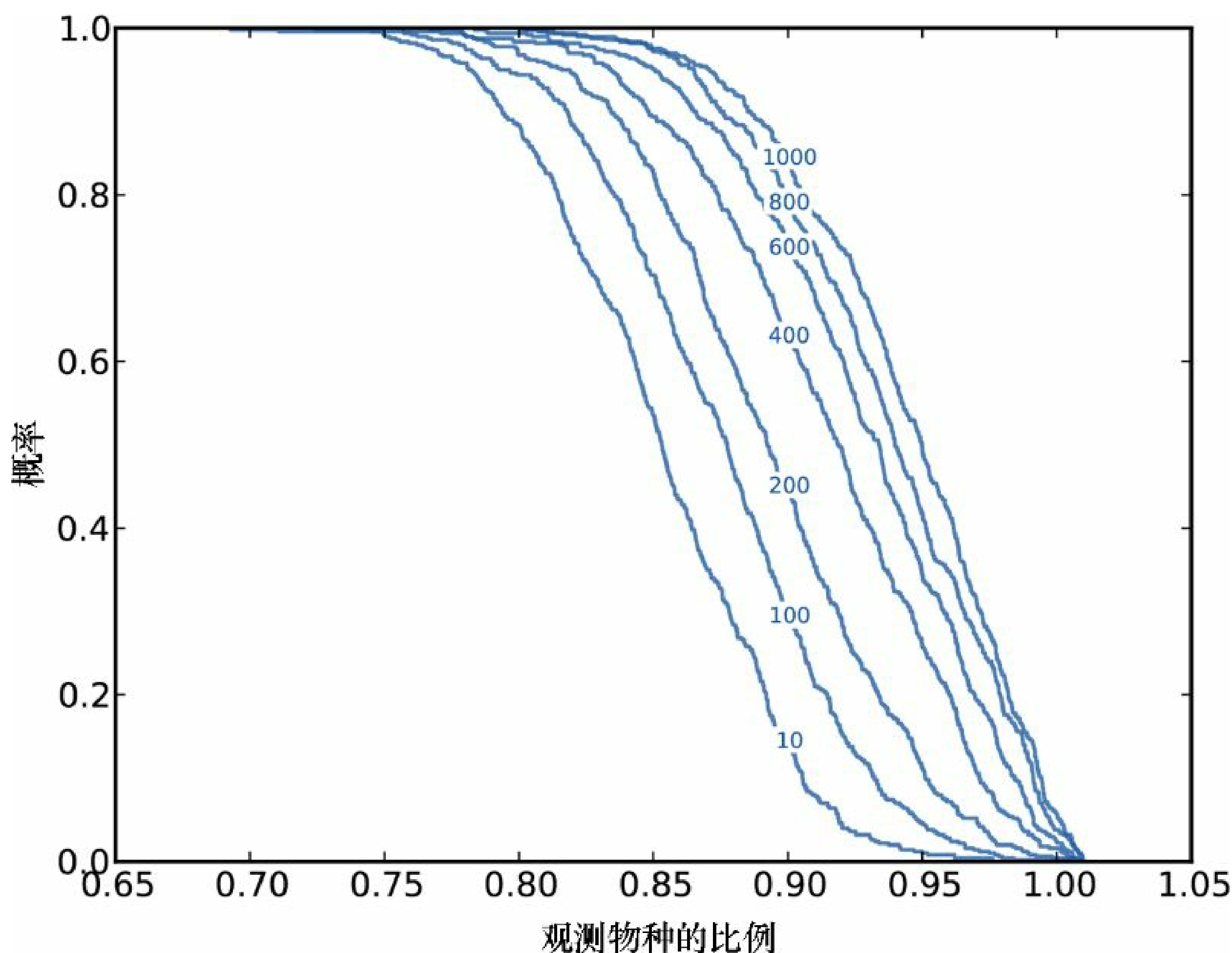


图15-7　额外标记样本数覆盖范围的互补CDF

要理解该图，沿着*x* 轴选取一个需要的覆盖率范围，例如90%，接着从图上就可以发现k 次样本标记后，要达到90%覆盖率的概率。例如，200次样本标记，有约40%的机会得到90%的覆盖率，而1000次样本标记，就有90%的机会得到90%的覆盖率。

如此，我们回答了未知物种问题的全部四个问题。要以实际数据验证本章中提到的算法，必须涉及更多细节，但是这章已经太长，所以这里我就不再继续讨论了。

你可以深入了解一下这类问题，以及我如何找到这些问题的，参考 *http://allendowney.blogspot.com/2013/05/belly-button-biodiversity-end-game.html* 。

你可以从 *http://thinkbayes.com/species.py.* 下载本章的代码，更多信

息请参考前言的“代码指南”。

15.13　讨论

未知物种问题是一个活跃的研究领域，我相信本章中的算法为此贡献了一些新意。在不到200页的篇幅里，我们已经从概率论基础扩展到了研究的前沿，这使我感到高兴。

本书中，我的目的是传达出3个关联的理念。

- 贝叶斯思维： 贝叶斯分析的立足点是使用概率分布来表示尚不明确的观点，通过数据来修正这些分布，使用得到的结果进行预测和提供决策信息。
- 计算方法： 本书的前提是使用计算方法而不是纯数学方法理解贝叶斯分析会更容易，通过可以组合且可以重复使用的程序框架来应用贝叶斯方法也更简单。
- 迭代模型 ：大多数真实问题都和建模决策以及于真实性和复杂性之间的权衡有关。预先知道哪些要素应该纳入模型，哪些要素可以被抽象到模型之外通常是不可能的。最好的办法就是迭代，从简单的模型开始再慢慢增加复杂度，使用每一个模型来交叉验证其他的模型。

这些观念实用且强大，从简单的例子到最新的研究课题，它们适用于科学研究和工程的每一个领域。

如果你领会到了，你应当已经为用这些工具来解决工作中的新问题做好了准备，希望你发现它的用处，要记得将你的收获告诉我。

---

① 译注：参考生物学文献，原文的“read”根据上下文翻译为标记样本或样本或样本标记。

作者简介

艾伦•唐尼 是欧林工程学院计算机科学系的教授。他曾在韦尔斯利学院、科尔比学院和加州大学伯克利分校教授计算机课程。

他拥有U.C.伯克利的计算机科学博士学位和麻省理工学院的硕士及学士学位。

译者简介

许杨毅，新浪网系统架构师，技术保障部总监，毕业于湖南大学，拥有15年互联网工作经验。

关于封面

《贝叶斯思维》一书封面的动物是红鲻鱼（也称为纵带羊鱼）。这种可以在地中海、北大西洋东海域和黑海发现的须鲷科鱼，因其第一背鳍后独特的条纹为人所知。红鲻鱼是地中海地区人们青睐的美味，和同属的须鲷科鱼——羊鱼一样，只是羊鱼没有第一背鳍后的条纹。然而红鲻鱼要更珍贵，据说其味道品尝起来类似生蚝。传说古罗马人在池塘中饲养红鲻鱼，宠爱并训练它们一听到钟声就喂饲。即便是人工养殖的红鲻鱼一般也不到两磅重，其价格有时和银器一样。

非野外环境下，红鲻鱼于浅底水系饲养，其上下唇具有独特的被称为触须的两根胡须，触须用来探测海底的食物。因为饲养于较浅的沙滩和岩石底部，它的触须与其深水的近亲羊鱼相比没有那么灵敏。

封面图片来自迈尔斯•克莱斯词典。

译后记

本书的翻译其实来自和编辑开过的一个玩笑，我当时戏说这本书的书名Think Bayes不妨译成《纪念贝叶斯老先生》，而实际上托马斯·贝叶斯先生也的确是一个伟大的值得我们纪念的人，科学的历史正是由这样一些伟大的人们推动的。贝叶斯发展并推出了我们现在熟悉的贝叶斯定理，而后人们也利用这一工具发展出了各种各样的“贝叶斯方法”。读完这本书，我相信读者对此会有深刻的认识。

另外就是本书的作者艾伦·唐尼。他不仅是一位计算机学科教授，同时也是一位优秀的作家。在翻译本书前，我已经读过他的Think Complexity 和Think Stats，在本书的翻译过程中，你还可以看到在github上他正在写作另一本关于操作系统的书，书名Think OS。

显然，Think Bayes是艾伦目前写得最好的一本书，原因在于在书中不但用编写Python程序的方式消除了贝叶斯方法的学习门槛，而且在每章的实际问题中，艾伦还潜移默化地教会了读者怎样为具体问题建立数学模型，如何抓住问题中的主要矛盾（模型中的关键参数），再一步一步地优化或者验证模型的有效性或者局限性。在这个过程中，你还能学习到统计分布中那些具体概念的实际含义和用途，比如边缘分布、联合分布、贝叶斯层次化模型。

作者艾伦·唐尼在github上托管了本书所有的Python代码，阅读本书的过程中，在自己的机器上运行例子中的代码绝对是一件让人满意的事情。

作者在写作中还带上了大量的概率图形，借助图表可以帮助读者更直观地理解有关问题的概率形式结论。

我的实际工作也和贝叶斯方法有关联，比如我们新浪邮箱就曾经采用贝叶斯方法处理垃圾邮件，业务监控系统也可以用贝叶斯方法来进行异常指标判断。至于谈到系统建模，本书也帮助我自己厘清了很多有关建模决策的困惑。

总之，这本不到200页的书绝对值得一读再读。

最后，我要感谢人民邮电出版社的编辑给我机会翻译艾伦的书。毫无疑问，艾伦是一个值得尊敬的科技书籍作者和良师。我还要感谢家人对翻译工作的支持，特别是我在书房静心工作的时候，刚刚一岁的小女儿波妞会很乖地和妈妈游戏，没有干扰我。谢谢我的女儿，我的爱人李静，当然还有无私地替我照顾妻女的母亲。

译者　许杨毅

2014年8月于颐和园北

欢迎来到异步社区！

异步社区的来历

异步社区(www.epubit.com.cn)是人民邮电出版社旗下IT专业图书旗舰社区，于2015年8月上线运营。

异步社区依托于人民邮电出版社20余年的IT专业优质出版资源和编辑策划团队，打造传统出版与电子出版和自出版结合、纸质书与电子书结合、传统印刷与POD按需印刷结合的出版平台，提供最新技术资讯，为作者和读者打造交流互动的平台。

社区里都有什么？

购买图书

我们出版的图书涵盖主流IT技术，在编程语言、Web技术、数据科学等领域有众多经典畅销图书。社区现已上线图书1000余种，电子书400多种，部分新书实现纸书、电子书同步出版。我们还会定期发布新书书讯。

下载资源

社区内提供随书附赠的资源，如书中的案例或程序源代码。

另外，社区还提供了大量的免费电子书，只要注册成为社区用户就可以免费下载。

与作译者互动

很多图书的作译者已经入驻社区，您可以关注他们，咨询技术问题；可以阅读不断更新的技术文章，听作译者和编辑畅聊好书背后有趣的故事；还可以参与社区的作者访谈栏目，向您关注的作者提出采访题目。

灵活优惠的购书

您可以方便地下单购买纸质图书或电子图书，纸质图书直接从人民邮电出版社书库发货，电子书提供多种阅读格式。

对于重磅新书，社区提供预售和新书首发服务，用户可以第一时间买到心仪的新书。

用户帐户中的积分可以用于购书优惠。100积分=1元，购买图书时，在 里填入可使用的积分数值，即可扣减相应金额。

> 特别优惠
>
> 购买本电子书的读者专享异步社区优惠券。使用方法：注册成为社区用户，在下单购书时输入“**57AWG**”，然后点击“使用优惠码”，即可享受电子书8折优惠（本优惠券只可使用一

次）。

纸电图书组合购买

社区独家提供纸质图书和电子书组合购买方式，价格优惠，一次购买，多种阅读选择。

软技能：代码之外的生存指南

[美]约翰 Z. 森梅兹（John Z. Sonmez）（作者） 王小刚 (译者) 杨海玲 (责任编辑)

分享 6 推荐 想读 9. 0K 阅读

这是一本真正从“人”（而非技术也非管理）的角度关注软件开发人员自身发展的书。书中论述的内容既涉及生活习惯，又包括思维方式，凸显技术中“人”的因素，全面讲解软件行业从业人员所需知道的所有“软技能”。

本书聚焦于软件开发人员生活的方方面面，从揭秘面试的流程到精耕细作出一份杀手级简历，从创建大受欢迎的博客到打造你的个人品牌，从提高自己工作效率到与如何与“拖延症”做斗争，甚至包括如何投资不动产，如何关注自己的健康。

本书共分为职业篇、自我营销篇、学习篇、生产力篇、理财篇、健身篇、精神篇等七篇，概括了软件行业从业人员所需的“软技能”。

纸质版 ¥59.00 ¥46.02（7.8 折）

电子版 ¥35.00

电子版 + 纸质版 ¥59.00

配套文件下载

现在购买

下载PDF样章

社区里还可以做什么？

提交勘误

您可以在图书页面下方提交勘误，每条勘误被确认后可以获得100积分。热心勘误的读者还有机会参与书稿的审校和翻译工作。

写作

社区提供基于Markdown的写作环境，喜欢写作的您可以在此一试身手，在社区里分享您的技术心得和读书体会，更可以体验自出版的乐趣，轻松实现出版的梦想。

如果成为社区认证作译者，还可以享受异步社区提供的作者专享特色服务。

会议活动早知道

您可以掌握IT圈的技术会议资讯，更有机会免费获赠大会门票。

加入异步

扫描任意二维码都能找到我们：



异步社区



微信订阅号



微信服务号



官方微博



QQ群：368449889

社区网址：www.epubit.com.cn

官方微信：异步社区

官方微博： @人邮异步社区，@人民邮电出版社-信息技术分社

投稿**&**咨询： contact@epubit.com.cn